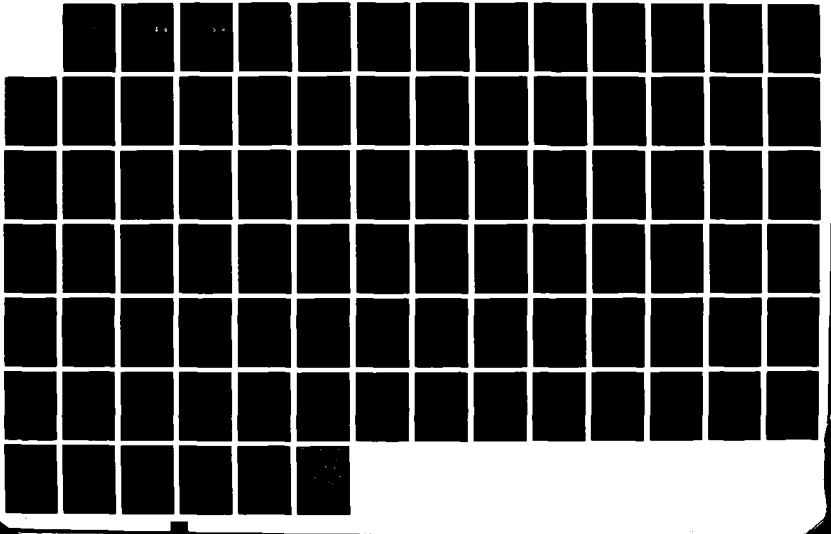
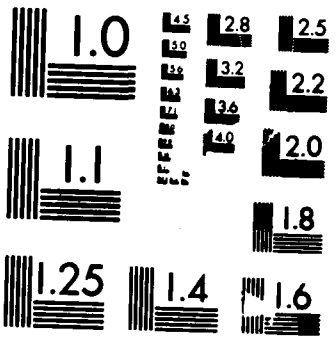
AD-A183 419 THE FORBIN PAPER(U) YALE UNIV NEW HAVEN CT DEPT OF 1/1
COMPUTER SCIENCE T DEAN ET AL. JUL 87 YALEU/CSD/RR-550
N00014-83-K-0281

UNCLASSIFIED F/G 21/1 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(6)

THE FORBIN PAPER

Thomas Dean, R. James Firby and David Miller

YALEU/CSD/RR #550

July 1987

YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

(6)

THE FORBIN PAPER

Thomas Dean, R. James Firby and David Miller

YALEU/CSD/RR #550

July 1987

This work was supported in part by Office of Naval Reasearch grants N00014-83-K-0281 and N00014-85-K-0301.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER #550	2. GOVT ACCESSION NO. AD-A183-419	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) The Forbin Paper		5. TYPE OF REPORT & PERIOD COVERED Research Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Thomas Dean, R. James Firby, David Miller		8. CONTRACT OR GRANT NUMBER(s) N00014-83-K-0281 N00014-85-K-0301
9. PERFORMING ORGANIZATION NAME AND ADDRESS Yale University - Dept. of Computer Science 51 Prospect Street New Haven, CT 06520		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22217		12. REPORT DATE July, 1987 13. NUMBER OF PAGES 72
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Program Arlington, VA 22217		15. SECURITY CLASS. (of this report) unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release: distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

-

18. SUPPLEMENTARY NOTES

-

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Task Networks
Scheduling
Temporal Reasoning
Hierarchical Planning

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Planning is the process of formulating sequences of actions which, if carried ou can be expected to bring about certain situations. Search in planning involves s ulating various sequences of actions to see whether or not they will bring abo desired situation. Since the space of possible sequences of actions is quite l comparison with the set of sequences that actually bring about the desire

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

tion, it is important to carefully guide the search. At each point in the planning process, the planner should have before it a representation that describes the planner's intentions and their potential consequences to help in deciding how to refine those intentions. This is the essence of what has been called *hierarchical planning*. The result of each decision constitutes a commitment on the part of the planner, and as commitments are made during planning they form the basis for subsequent decisions. The perfect planner would never have to retract or reconsider its previous commitments. But then the perfect planner wouldn't really have to plan; it would simply know, instinctively as it were, what to do next. In most situations, it is impossible to guarantee that a decision, once made, is immune to retraction. In the sort of routine planning situations that we are concerned with, retracting previous commitments, or *backtracking* as it is commonly referred to, while it can't be avoided altogether, can be carefully controlled. Backtracking, rather broadly construed as "trying another alternative", is essentially synonymous with search, and, as such, it is unavoidable. One can, however, direct the decision making process so that, having performed a certain amount of search (significantly less than that required for the entire problem), one can reliably commit to the consequences of a particular decision without fear of later retraction. This paper describes a planning architecture that supports a form of hierarchical planning well suited to applications involving deadlines, travel time, and resource considerations.

OFFICIAL DISTRIBUTION LIST

Defense Documentation Center — 12 copies
Cameron Station
Alexandria, Virginia 22314

Office of Naval Research — 2 copies
Information Systems Program
Code 437
Arlington, Virginia 22217

Dr. Judith Daly — 3 copies
Advanced Research Projects Agency
Cybernetics Technology Office
1400 Wilson Boulevard
Arlington, Virginia 22209

Office of Naval Research — 1 copy
Branch Office - Boston
495 Summer Street
Boston, Massachusetts 02210

Office of Naval Research — 1 copy
Branch Office - Chicago
536 South Clark Street
Chicago, Illinois 60615

Office of Naval Research — 1 copy
Branch Office - Pasadena
1030 East Green Street
Pasadena, California 91106

Mr. Steven Wong — 1 copy
New York Area Office
715 Broadway - 5th Floor
New York, New York 10003

Naval Research Laboratory — 6 copies
Technical Information Division
Code 2627
Washington, D.C. 20375

Dr. A.L. Slafkosky — 1 copy
Commandant of the Marine Corps
Code RD-1
Washington, D.C. 20380

Office of Naval Research — 1 copy
Code 455
Arlington, Virginia 22217

Office of Naval Research — 1 copy
Code 458
Arlington, Virginia 22217

Naval Electronics Laboratory Center — 1 copy
Advanced Software Technology Division
Code 5200
San Diego, California 92152

Mr. E.H. Gleissner — 1 copy
Naval Ship Research and Development
Computation and Mathematics Department
Bethesda, Maryland 20084

Captain Grace M. Hopper, USNR — 1 copy
Naval Data Automation Command, Code 00H
Washington Navy Yard
Washington, D.C. 20374

Dr. Robert Engelmore — 2 copies
Advanced Research Project Agency
Information Processing Techniques
1400 Wilson Boulevard
Arlington, Virginia 22209

Professor Omar Wing — 1 copy
Columbia University in the City of New York
Department of Electrical Engineering and
Computer Science
New York, New York 10027

Office of Naval Research — 1 copy
Assistant Chief for Technology
Code 200
Arlington, Virginia 22217

Computer Systems Management, Inc. — 5 copies
1300 Wilson Boulevard, Suite 102
Arlington, Virginia 22209

Ms. Robin Dillard — 1 copy
Naval Ocean Systems Center
C2 Information Processing Branch (Code 8242)
271 Catalina Boulevard
San Diego, California 92152

Dr. William Woods — 1 copy
BBN
50 Moulton Street
Cambridge, MA 02138

Professor Van Dam
Dept. of Computer Science
Brown University
Providence, RI 02912
1 copy

Professor Eugene Charniak
Dept. of Computer Science
Brown University
Providence, RI 02912
1 copy

Professor Robert Wilensky
Univ. of California
Elec. Engr. and Computer Science
Berkeley, CA 94707
1 copy

Professor Allen Newell
Dept. of Computer Science
Carnegie-Mellon University
Schenley Park
Pittsburgh, PA 15213
1 copy

Professor David Waltz
Univ. of Ill at Urbana-Champaign
Coordinated Science Lab
Urbana, IL 61801
1 copy

Professor Patrick Winston
MIT
545 Technology Square
Cambridge, MA 02139
1 copy

Professor Marvin Minsky
MIT
545 Technology Square
Cambridge, MA 02139
1 copy

Professor Negroponte
MIT
545 Technology Square
Cambridge, MA 02139
1 copy

Professor Jerome Feldman
Univ. of Rochester
Dept. of Computer Science
Rochester, NY 14627
1 copy

Dr. Nils Nilsson
Stanford Research Institute
Menlo Park, CA 94025
1 copy

The Forbin Paper

Thomas Dean
Brown University
P.O. Box 1910
Providence, RI 02912

R. James Firby
Yale University
P.O. Box 2158 Yale Station
New Haven, CT 06520

David Miller
Virginia Polytechnic Institute
Blacksburg, VA 24061

YALEU/CSD/RR #550

July 1987

Much of the work described in this paper was carried out while all three of the authors were together at Yale University. The order of names is alphabetical and has no other significance.

Abstract

Planning is the process of formulating sequences of actions which, if carried out, can be expected to bring about certain situations. Search in planning involves simulating various sequences of actions to see whether or not they will bring about the desired situation. Since the space of possible sequences of actions is quite large in comparison with the set of sequences that actually bring about the desired situation, it is important to carefully guide the search. At each point in the planning process, the planner should have before it a representation that describes the planner's intentions and their potential consequences to help in deciding how to refine those intentions. This is the essence of what has been called *hierarchical planning*. The result of each decision constitutes a commitment on the part of the planner, and as commitments are made during planning they form the basis for subsequent decisions. The perfect planner would never have to retract or reconsider its previous commitments. But then the perfect planner wouldn't really have to plan; it would simply know, instinctively as it were, what to do next. In most situations, it is impossible to guarantee that a decision, once made, is immune to retraction. In the sort of routine planning situations that we are concerned with, retracting previous commitments, or *backtracking* as it is commonly referred to, while it can't be avoided altogether, can be carefully controlled. Backtracking, rather broadly construed as "trying another alternative", is essentially synonymous with search, and, as such, it is unavoidable. One can, however, direct the decision making process so that, having performed a certain amount of search (significantly less than that required for the entire problem), one can reliably commit to the consequences of a particular decision without fear of later retraction. This paper describes a planning architecture that supports a form of hierarchical planning well suited to applications involving deadlines, travel time, and resource considerations.

Acknowledgements

This paper was a long time in the writing and would not have been possible at all without the patience and support of Drew McDermott. The clarity of the paper, such as it is, also owes much to long discussions with Yoav Shoham and Steve Hanks. We would like to thank these three people and the many others whom we have harrased over the last two years.

This report describes work done by all three authors at Yale University supported in part by the Advanced Research and Projects Agency of the Department of Defense under Office of Naval Research contract N00014-83-K-0281. Subsequent discussion and writing was continued after two authors moved on and was further supported in part by DARPA Office of Naval Research contract N00014-85-K-0301 at Yale University, NSWC contract N60921-83-G-A165 at Virginia Polytechnic and NSF grant IRI-8612644 at Brown University. In addition, Thomas Dean was supported at Brown University through an IBM Faculty Development Award and Jim Firby was supported at Yale University through a Canadian NSERC 1967 Science and Engineering Scholarship.

Contents

List of Figures

List of Tables

1 Introduction

Programming robots and automatic machines to perform complex tasks is a difficult and tedious operation. In a domain of any complexity, it would be difficult if not impossible to construct a set of plans to cover every possible contingency. One way around this problem is to program the robot so that when assigned a new task it can construct its own detailed plan to suit the particular circumstances.

There has been a great deal of work done by AI researchers in the field of automatic planning. Unfortunately the word "planning" often means different things to different people. Since this paper discusses many of the issues that arise when constructing planners and in fact describes an existing planner in some detail, it is important to make clear just what we mean by planning. Planning is the process of formulating sequences of actions which, if carried out, can be predicted to bring about certain desired situations. An automatic planner is given the description of a task specifying certain facts that should be made true and is expected to generate from this specification, and the planner's general knowledge of a particular domain, a sequence of actions that will serve to make those facts true. In most planning situations the problem is not merely one of finding a sequence of actions that works, but finding a sequence that works well, where "well" usually means something like "can be completed in a short amount of time" or "with a small expenditure of resources."

In the rest of this section, we will consider what this process of formulating sequences of actions actually entails. In doing so, we will introduce a number of themes that will run throughout the paper.

1.1 Planning as Search

Planning is generally viewed as a process of incremental construction. At any given point in the process, the planner has before it a partially constructed plan that it is working on. A partially constructed plan corresponds to a set of commitments to act in various ways, where the actions or the order in which they are to be executed may not be completely specified. For instance in constructing a plan to manufacture a particular part, a planner may commit to using a specific manufacturing process without committing to a specific machine with which to carry out the process. Planning proceeds as a series of decisions concerning extensions to the current set

of commitments. The new commitments serve to refine (or provide more detail to) the partially constructed plan. The decisions required of a planner define a search space in which each node corresponds to a partially constructed plan and each arc corresponds to an additional commitment refining a partially constructed plan.

In order to guide search, the planner has to have some means of assessing its progress. Such guidance is usually provided by some sort of mechanism for validating or evaluating partially constructed plans. Validation involves predicting the consequences of the actions committed to thus far and then determining that these actions are likely to bring about the desired state of affairs. There is no way to completely validate a partially constructed plan without actually planning out all the details but, possessed of appropriate knowledge, a planner can reason about actions and their effects at many different levels of detail. Given such knowledge, validation corresponds to predicting that the partially constructed plan brings about the desired state of affairs given the current level of detail.

1.2 Knowledge Required for Planning

The amount of knowledge required for planning in any nontrivial domain is immense. In order to evaluate its progress in constructing a plan, a planner has to be able to reason about the consequences of events outside its control as well as the effects of its own actions. In order to efficiently guide the planning process, a planner must possess knowledge about how to proceed in constructing a plan (*i.e.*, how to refine a partially constructed plan). In addition to this largely static knowledge, the planner has to keep track of its commitments (*i.e.*, the partially constructed plan) and its predictions concerning its own actions and the consequences of events outside its control.

We can divide the planner's static knowledge into two components: a database of rules that establish the cause-and-effect relations of a particular domain and a database of rules that establish the conditions for using certain methods for refining partially constructed plans. The planner's changing knowledge concerning its partially constructed plan and the observed and predicted events corresponding to the context in which this plan is to be executed is captured in a temporal database. The planner uses this temporal database and its rules for refining partially constructed plans to generate additional commitments. These new commitments are

then used to update the temporal database to reflect the consequences predicted by the system's causal knowledge.

1.3 Decision Making

The process of refining a partially constructed plan is usually referred to as *hierarchical planning.* The main idea behind hierarchical planning is that the refinement process can be staged in such a way that decisions made early in planning are independent of decisions made later. It has traditionally been assumed that refinement decisions can be made by referring only to the current partially constructed plan. In particular, it is generally assumed that if the actions in the partially constructed plan are only partially ordered, it is not necessary to consider any of the possible orders consistent with this partial order. Unfortunately, this assumption is not warranted in most domains. If we are considering travel time or renewable resources, the order in which actions are carried out is critical. The planner described in this paper makes decisions based upon the predictions generated by two programs. The first program is responsible for keeping track of the big picture; it operates upon the entire contents of the temporal database and reasons directly about partial orders without considering restrictions on the current partial order. The second program is generally assumed to operate upon a subset of the facts stored in the temporal database; it makes predictions based upon considering a subset of all total orders consistent with the current partially constructed plan. The first program is incomplete and often imprecise, but it attempts to provide a global perspective. The second program, by concentrating upon a smaller data set, is able to be very precise, but it gains this precision at the expense of considering a much smaller set of possibilities. Exactly how these two programs are used in deciding what refinements to make to a partially constructed plan is central to this paper.

1.4 Relation to Previous Work

A large amount of work has been done on the problem of building a general purpose hierarchical planner [3] [4] [14] [15] [16] and two of the early, influential systems are NOAH [14] and NONLIN [15]. These systems use a partially ordered network of tasks to represent the plan at each level of expansion and they employ a strategy called *least commitment* that delays adding new ordering constraints as long as possible.

The notions of a partial order and least commitment have become central to all planners because they decrease the need to backtrack and undo previous expansion decisions. However, neither NOAH nor NONLIN represent time or continuous change (like robot travel) in an adequate manner and they are thus incapable of solving planning problems where these features are crucial.

The DEVISER system [16] extends hierarchical planning to include time and continuous change. DEVISER, however, does not exploit the temporal information it represents to guide the planning process as much as it could. Early expansion and ordering choices are made arbitrarily and only later, when the plan is expanded to great detail, is temporal information used to rule out certain planning possibilities. To tackle a wide class of problems in this way, DEVISER must rely on backtracking to fix early mistaken choices. Another planning system that uses time, called ISIS [9], is designed to efficiently schedule the distribution and progress of tasks in a large machine shop. Unfortunately, the ISIS system is extremely specialized to the job-shop scheduling domain and cannot be considered a solution to the general class of planning problems having to do with resources, travel time, and continuous change.

This short subsection is only meant to establish a few connections that should be familiar to many readers. We will have more to say about competing theories in the course of expounding upon our own.

1.5 About This Paper

This paper describes the FORBIN planner: a system for generating plans in domains involving mobile robots, deadlines, and limited resources. We will describe the FORBIN planner in some detail. We will also describe what we learned in building FORBIN system. It is the latter that we believe to be more important, but obviously if the reader is to benefit from what we learned, we will have to provide an appropriate context. Before we begin describing our work, there are a few things that we would like the reader to be aware of.

1.5.1 FORBIN: The Program

It is no accident that the name FORBIN (as in "The Forbin Project") is intimately connected with a fictional computer called "The Colossus" brought to life in a movie of the same name. FORBIN is quite a large program as planning systems go (on the

order of 30K lines of LISP code). Unlike its namesake however, FORBIN is not likely to pose any threat to the safety of the world; not, at least, in the foreseeable future. FORBIN is interminably slow. The program is slow because it tries to do too much too well. The FORBIN planner was conceived with two seemingly innocuous design criteria in mind. We wanted to build a planner for a nontrivial domain that would (i) find a plan where one exists, and (ii) assuming that a plan does exist, find a good plan. The main problem is that FORBIN attempts to solve (or at least approximately solve) problems that are inherently combinatoric (*i.e.*, problems that lack the necessary structure to enable them to be efficiently solved). In order to solve the problems it was designed for, FORBIN had to sacrifice performance (speed) or completeness. Each time one of us suggested introducing some feature that contributed to FORBIN being incomplete, the rest of us would hesitate, realizing that if we allowed this feature there would be no way of characterizing the conditions under which FORBIN would succeed. In the end, we reluctantly introduced several sources of incompleteness into the FORBIN algorithms. At the same time, we developed a complicated architecture that attempted to ensure that this incompleteness was minimized. Completeness and performance were continually fighting for preeminence, and performance was more often than not the loser. We tried desperately to shore up our foundering colossus by adding all sorts of heuristics. It is our contention, however, that, in the end, the domain-dependent knowledge necessary for efficient planning will overshadow the so-called domain-independent knowledge about planning as well as the sort of architectural considerations that have dominated the planning literature over the last decade.

To be fair, FORBIN was a working program, and it was used to conduct a small number of valuable experiments. All of the primary components of FORBIN have been tested and run in other planning systems, and all of the components ran together in the FORBIN architecture. FORBIN would quite likely have been able to handle reasonably sized plans had its developers not been forced to do their development on an underconfigured VAX-750. As it was, FORBIN was actually run on very simple tasks or used to partially construct plans for more complex tasks. Perhaps if we had had access to larger machines, we might have been able to fool ourselves that FORBIN was a viable approach to planning, but eventually the combinatorial explosion would have caught up with our experimentation. Believing ourselves to be as competent as other (seemingly) more successful researchers, we suspect that many planning systems suffer from the same ills that crippled FORBIN.

but that the researchers responsible for building those systems simply have not considered problems of the complexity that FORBIN attempts. The rest of this paper argues that if traditional planning in nontrivial domains has a chance of succeeding, then the design of FORBIN should serve as a model for future planners. In presenting our argument, we will also indicate specific places in our algorithm where detailed knowledge of the domain is a necessary prerequisite to reasonable performance.

1.5.2 FORBIN: The Paper

There are a number of things that might be learned from reading this paper. Sections 1 and 2 provide a self-contained overview of a particular paradigm of planning that has engaged a fair number of researchers over the last two decades. These sections might serve as an introduction to the subject for someone without a great deal of familiarity with planning issues. Sections 2, 3, and 7 provide a fairly high-level description of our particular contribution to this paradigm: what we did and why we think it is important. The basic ideas are set out in Sections 2 and 3; Section 7 summarizes these ideas and places them in perspective. Sections 4, 5, and 6 provide the details necessary to understand some of the more esoteric contributions of FORBIN, such as the representational contributions concerning reasoning about plans, time, and causality. Section 4 is concerned with the details of causal theories, Section 5 describes the language for specifying plans, and Section 6 provides a detailed example illustrating how FORBIN actually works. Throughout this paper we will use examples drawn from an automated factory domain. Most of the examples do not require detailed knowledge of the domain. Appendix A provides the details for those interested.

2 Planning as Search in the FORBIN System

The central problem in planning research is to manage the amount of search done in building a reasonable plan. An automatic planning program spends the majority of its time searching through the space of all plans that it could generate in order to find a good plan. The FORBIN planner is no different. For a typical task assigned to FORBIN, there will be a large number of different ways to carry it out. FORBIN's job is to find one plan that will carry it out well. This section of the paper describes the basic design decisions involved in choosing the search methods used by the FORBIN program.

2.1 Planning Search Primitives

The fundamental primitive, or *operator*, in the FORBIN planning search space is the ***simple action***. Every planner abstracts differently from the details of its specific domain, and, hence, what is considered simple will depend on a particular planner. When you ask for a plan to build a house, you are probably not interested in the details of hammering nails, but neither are you likely to be satisfied with an exhortation to consult a contractor or buy a kit and assemble it. An action is simple when breaking it down into even simpler actions would reveal no additional useful information given the target level of abstraction of the planner. Examples of simple actions in the FORBIN domain are "Pick up the widget" and "Push the lathe start button".

A task corresponds to a commitment on the part of the planner to carry out a certain action. Typically, the action specified in a task is not simple; it is said to be ***problematic***. Most planners deal with tasks that involve making certain facts true of the world. For example, if the desired fact is that block `A` is on block `B` (*i.e.*, `(ON A B)`), then the associated task is `(ACHIEVE (ON A B))`. Carrying out such a task corresponds to executing primitive actions that bring about a state of the world in which the desired fact is true. Using this notion of task, planning is defined as the search for a sequence of simple actions that will carry out all of the tasks that the planner has been assigned. In all planning systems, this search begins with the planner knowing in detail the state that the world will be in when the first step in the plan is executed. Each simple action is represented as an operator that will transform the world from one state to another when actually executed. Thus,

finding a plan to carry out a set of tasks is equivalent to searching for a set of operators that will transform the initial world state into a new state in which all of the facts associated with the planner's tasks are true. To perform this sort of search, the planner has to be able to determine what the state of the world is just prior to applying a given operator, and it has to know exactly how each operator changes that state. The more things that can be represented in the world state and the more ways that operators can change that state, the wider the class of problems that the planner can solve. However, as the complexity of the planner's state representation increases, so does the complexity of the search it must perform. As planning domains become more realistic, the search strategies required to deal with them are shaped more and more for effective management of complexity than for simply getting the right answer. Getting an answer is not good enough if it takes too long to compute.

2.2 Search Strategies

The simplest search strategies are based on the idea of chaining operators together one after another until an appropriate world state is produced. Forward chaining starts with the initial world state and adds operators to move the state towards a final state where all tasks have been accomplished. Backward chaining starts with the final state and adds operators in reverse order until the initial world state is produced. One of the earliest problem solvers, GPS [7], uses a search scheme combining forward and backward chaining and many subsequent planners (*e.g.*, STRIPS [8]) have followed that lead. In chaining search schemes, each simple action is represented as an operator annotated with preconditions and effects. The preconditions describe states of the world where executing the simple action is appropriate and the effects describe changes to that state caused by its execution. The plan is grown forward by adding an operator with satisfied preconditions and computing the following state from its effects, or backward by adding an operator with satisfied effects and computing the previous state from its preconditions. The state of the search is captured completely in the sequence of operators applied thus far and a description of the state of the world following or preceding that sequence of operators (depending on whether the search method involves forward or backward chaining). When more than one operator can be added, some form of utility function is required to guess which one will make a better final plan. If a state is ever reached where no

operator can be added, the planner backtracks and selects the next best operator at an earlier choice.

This search strategy is attractive in its simplicity but its performance can be extremely poor. In the case where the utility function selects the best choice at every step, the search algorithm will only have to consider a single plan, but, if the utility function is poor, the planner may have to consider every possible combination of operators before finding a plan that works. Indeed, there may be sequences of operators that are infinitely long and the search would never terminate. In realistic domains where plans at the level of simple actions are usually quite long and where many actions are applicable in any state, one requires an extremely good utility function if the search is to remain tractable. Research in planning has so far failed to turn up a general theory for building utility functions and indications are that any good function will be highly dependent on the types of tasks the planner is designed to solve.

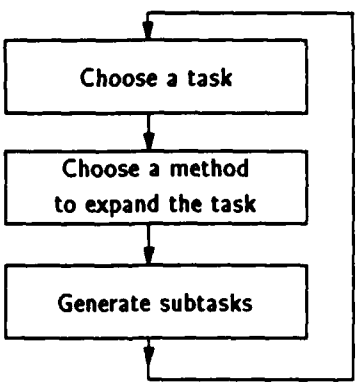


Figure 1: Basic algorithm for task expansion

To build a general purpose planning system, reliance on task dependent utility functions must be reduced. One way to do this is to alter the basic search strategy from simple chaining to *hierarchical expansion*. In a hierarchical expansion planner, the operators for simple actions are grouped into methods for performing more general actions. These general actions are in turn represented by operators that can

be used to build even more general methods. The result is that the planner ends up with a library of hierarchically defined abstract operators at its disposal. Each abstract operator is associated with one or more methods, and the methods are used to *expand* an abstract operator into a set of less abstract operators. Expansion is equivalent to generating new tasks (or subtasks), since, by committing to a particular method, the planner commits to the abstract operators (think of them in terms of abstract actions) stipulated by the method. When assigned a set of tasks to accomplish, the planner looks up the abstract operator for each task and chooses an appropriate method for carrying it out. Eventually all of the abstract operators will be expanded into simple actions and the plan will be complete. The basic cycle of activity is depicted in Figure 1. This method of hierarchical expansion can result in better performance than methods relying on simple chaining because the set of possible plans for a given problem is restricted to those implicitly defined by expansion methods in the library. To achieve reasonable performance, hierarchical planning trades a task dependent utility function for task dependent expansion methods in its library. The hope is that such methods will be more widely applicable and easier to develop than the corresponding utility functions.

In the above way of thinking about planning, the initial set of tasks corresponds to a partially constructed plan, albeit a plan at a fairly high level of abstraction. Each task is associated with an interval of time during which the task is to be carried out and generally there are constraints on the order and completion time of tasks. At all times the planner has a partially constructed plan that it is working on. This partially constructed plan is represented in what is called a *task network* [10]. Figure 2.a illustrates a simple task network corresponding to two initially supplied tasks. Expansion results in adding subtask and precedence links to the network. Figure 2.b shows the result of expanding each of the two tasks in Figure 2.a. The task network encodes the basic commitments of the planner. As we will see, there is a great deal more that has to be represented to support hierarchical planning. In particular, the consequences of the planner's proposed actions as well as the consequences of events outside the planner's control all have to be taken into account in evaluating a partially constructed plan.

Figure 3 illustrates the basic architecture of planners that perform hierarchical expansion. Arrows indicate the flow of information between modules, circles represent static knowledge sources, and the box labeled "temporal database" indicates the dynamic component of the planner's knowledge. The task expander implements the

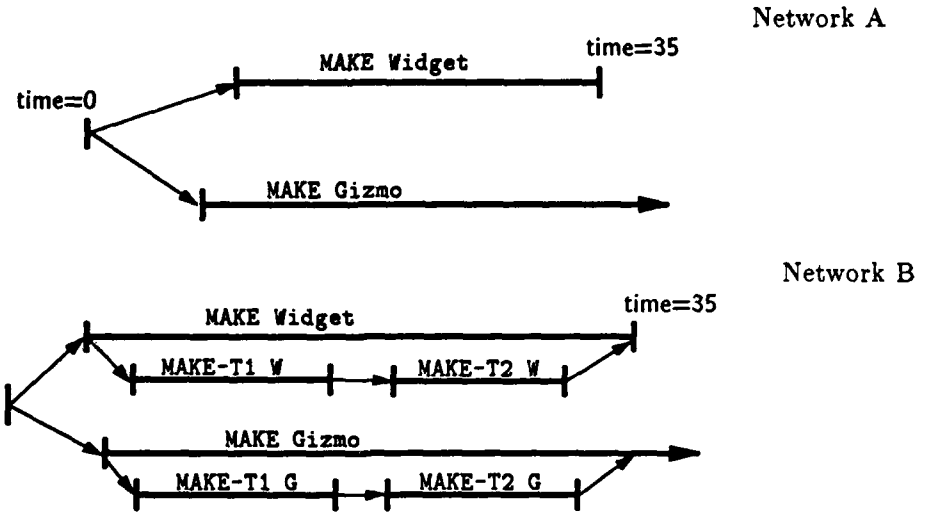


Figure 2: Simple task networks

algorithm sketched in Figure 1, and the query processor assists in making choices by answering questions concerning the contents of the temporal database. Planners can be distinguished by the expressivity of the languages used for representing plans and causal rules and by the techniques used for reasoning about such plans and causal rules. The temporal database contains a set of cached deductions concerning the state of the partially constructed plan and (some of) the consequences of the proposed actions. In the NASL planner [10], the temporal database is no more than the task network itself. In other planners, the temporal database may consist of a task network annotated with information concerning the truth of propositions corresponding to the effects of actions (*e.g.*, the *table of multiple effects* of NOAH [14]).

The main difficulty with hierarchical planning is sorting out unexpected interactions between the expansion methods for different operators. The operators and the order of their application within a single method will have been chosen so that they do not interfere with each other and success will be assured at execution time. However, when a plan is built for several initially independent high-level tasks,

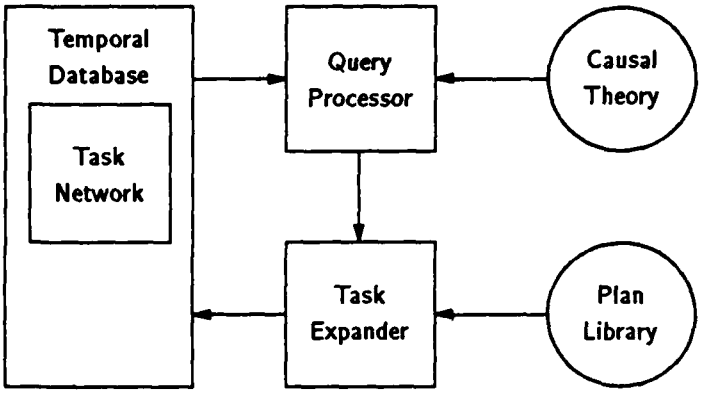


Figure 3: Basic architecture for hierarhical planning

operators from the different expansions may become interspersed and cause interactions that compromise the previously assured success. To notice and deal with such interactions, the planner must keep track of the expected state of the world generated by each operator, both simple and abstract, in the plan it has built so far. As with chaining algorithms, this requires that each operator be annotated with the world state that it requires before it can be applied and with the way that it changes that state when executed. By choosing expansion methods and orderings that preserve the world states required by operators already chosen, bug free plans with interspersed operators can be produced. When a bug is found, the planner can backtrack and choose a different method for an earlier expansion. Although monitoring the world state throughout the plan increases the cost of the planning search, the hierarchical nature of the plan means that all expansion choices are made in the context of the whole future at some level of detail. Again, the hope is that this context will generally help the planner zero in on a good plan with little or no backtracking and hence little search.

2.3 Summary

Hierarchical expansion as a search method is a way of drastically pruning the space of plans that might be considered for a task. Hierarchical expansion using a library

of known methods for each type of task ensures that even partial plans considered during the search will form complete solutions at some level of detail and that such solutions will be good, relevant candidates. In fact, one might consider the possibility that given only a single task to perform, hierarchical expansion would lead to consideration of only a single plan with no real search at all. Real planning domains however, inevitably allow more than one way for expanding a task and usually demand interleaving subtasks from separate task expansions to avoid gross inefficiences. These considerations lead to the need for choosing between alternative expansion methods and for detecting undesirable interactions between expansion methods. To evaluate different expansion options and detect interactions the FORBIN planner must continually project the expected future at each point in its plan. Thus, within the FORBIN system, the problem of planning splits into two major subproblems: building a plan via search through its expansion hierarchy and monitoring the evolving plan for consistency and efficiency via temporal projection.

3 An Overview of the Planner and its Algorithm

Although the discussion in the previous section makes it clear that the FORBIN planning problem can be broken up into expansion and projection, the real contributions of the FORBIN system center around exactly how this is done. In the introduction, we mentioned that the planner's static knowledge can be thought of as residing in two separate databases: one containing rules concerning cause-and-effect relations and one containing rules about how to refine partially constructed plans. In addition, FORBIN makes use of a temporal database to keep track of the plan it is incrementally constructing. The temporal database is also used to track other known events and the consequences of those events and the planner's proposed actions as predicted by the causal rules. This database is updated via temporal projection each time the planner commits to additional detail in the plan it is constructing.

In choosing how to refine a partially constructed plan, the planner consults the database of refinement methods looking for suitable methods. It then uses the current state of the temporal database to determine if any of those methods are suitable in the current circumstances. At this point the planner generally has a small number of candidate methods for refining its partially constructed plan. To choose between these methods, the system performs some additional very detailed projections in an effort to find the best method for refining its plan. The only backtracking that FORBIN engages in is buried in the routines responsible for performing these detailed projections. Once it has chosen a refinement method, the method is applied to the current plan and the temporal database is updated accordingly. If FORBIN ever finds itself unable to find an appropriate refinement method, it simply stops and signals failure. Though FORBIN employs very sophisticated routines for tracking the reasons why it made certain decisions, FORBIN currently has no theory about how to recover from inappropriate decisions. As we will see, however, FORBIN goes to great lengths to make a good decision the first time around.

Before we describe the FORBIN algorithm in more detail, we will briefly consider some of the primary components of the reasoning process.

3.1 The Causal Theory

The causal theory specifies the way the world changes both as a result of planned actions and as a result of autonomous processes. In particular, the theory contains

rules that determine the effects that occur as a result of performing certain actions. The causal theory does not distinguish between intended effects and those that occur as side effects of purposefully executed actions. Most effects correspond to simple facts that are made true as a result of actions being executed or, more generally, events occurring. But other effects deal with quantities that change continuously over time: quantities the planner can add to, subtract from, or begin changing by starting a process that will continue the change on its own. For example, the action of starting an automatic milling machine has the simple effect of changing its state from off to on and the more complex effects of using up raw material to make the part, causing wear on the cutting bit, and at some point in the future causing a finished part to exist where there was none before. A useful causal theory must concisely capture the interactions that occur between events and their associated effects. Turning on a light might cause a solar powered sculpture to begin turning, but there should be no mention of solar powered sculptures in the causal rules describing the effects of toggling light switches. Rather, the connection between the switch and sculpture should be encoded in a set of rules that refer to much more general physical theories.

3.2 The Temporal Database

The temporal database is used by the planning system to maintain a picture of the world changing over time in accord with the current causal theory, the planner's anticipated actions, and known autonomous processes. When planning begins, the temporal database is given the initial world state and the time and nature of any events that will take place in the world outside of the planner's control. The database update routines then apply the rules from the causal theory to obtain a concise record of the state of the world at all times in the future. As the plan is constructed, each action the system commits to as part of the plan is added to the database along with the time it is expected to occur. This time may be with respect to a global time frame (*e.g.*, between 3:00 and 3:15 PM on Tuesday) or it may be with respect to the time frame of another event (*e.g.*, after turning on the lathe). In either case, the database again applies the rules from the causal theory and recomputes the world state at all times in the future, now including changes anticipated from the new action. Thus the database constantly maintains an accurate picture of the future to be expected when the plan under construction is executed.

In addition to keeping track of facts that are true in the world, the database protects the truth of facts that need to remain true for the plan being built to remain appropriate. Usually the choice of a particular action to add to the plan will depend on the situation expected to exist at the time the action is to be executed. For example, the planner might decide that the best way to get a new widget by 2:00 PM is to make it on the lathe and add such an action to the database. During later planning however, the planner might decide that to fill an important order, it would like to use the lathe to make gizmos all day long. Simply adding this additional use of the lathe to the database would lead to an inconsistent view of the future with the planner thinking it could do both actions at the same time. To avoid this situation two things are done. First, when an action is added to the database, facts assumed true in choosing the action are recorded along with the action. Second, before a new action is chosen for use in the plan, the system asks the database whether adding it will cause any previous assumptions to be undone. Inconsistent futures are avoided by never adding an action to the database that will violate the assumptions of an action added previously.

3.3 The Task Expander

A planning problem is specified to the planner as a set of abstract actions and constraints on when these actions should be executed. The abstract actions are generally referred to as *tasks*. Refining a partially constructed plan proceeds by expanding a task into a set of simpler subtasks until only primitive actions requiring no further refinement remain. We refer collectively to the routines responsible for performing these expansions as the *task expander*. The database of refinement methods consists of alternative ways of expanding various types of tasks into subtasks. The methods are arranged hierarchically by type where the hierarchical ordering is in terms of the level of detail of the resulting subtasks. Refinement in FORBIN usually proceeds through several levels of abstraction culminating in expansions involving only primitive actions. Hierarchical refinement is used so that the temporal database always contains a view of the whole future at some level of detail. Having the whole future represented in the database provides the basis for the next level of expansions and allows FORBIN to take into account important prerequisites of tasks early in planning.

3.4 The Basic FORBIN Algorithm

The basic planning algorithm is easy to understand in terms of the task expander, the temporal database, and the causal theory. The task expander selects an abstract task from the database and looks up all appropriate refinement strategies in the expansion library. It then asks the database which expansion strategies are appropriate. If no strategies will work then the task cannot be expanded and the planner must fail in its attempt to build a finished plan. If one or more strategies will work, then the task expander chooses the "best". Using the best strategy found, the chosen task is expanded, added to the temporal database, and the predictions recorded in the database are updated in accordance with the causal theory. This cycle of choosing a task to work on, picking the best expansion based on the expected future and then updating the temporal database continues until either some task cannot be expanded and failure occurs or all tasks have been reduced to primitive actions and the plan is complete.

The previous paragraph describes the FORBIN planning algorithm in a fairly superficial manner. In order to explain how FORBIN attempts to avoid backtracking, a little more detail is required. FORBIN was designed to operate in complex domains and cope with large numbers of tasks. It was generally assumed that the temporal database would contain on the order of thousands of entries corresponding to tasks at various levels in the abstraction hierarchy. These tasks would be partially ordered and have a multitude of possible consequences. The task of accurately projecting all of the consequences of these tasks was judged to be impossible[1]. Instead FORBIN was designed to rely upon a projection algorithm that was incomplete (at least in dealing with partially ordered events) but polynomial in the number of tasks in the temporal database. The FORBIN procedures used in choosing an initial set of expansion strategies were not guaranteed to be correct, but they were guaranteed to be fast (for details see [5]). To compensate for these incomplete procedures, FORBIN employed an additional set of procedures for projecting consequences and reasoning about partial orders. These procedures were able to score expansion strategies by actually performing the expansions in a temporary database and then simulating the resulting partial plans by considering various total orders consistent with the current partially constructed plan. Scoring was handled using domain-specific util-

[1]The general problem of projecting the consequences of a partially ordered set of tasks using a causal theory capable of representing actions whose effects depend upon the context in which they occur is *NP-hard* [3].

ity functions encoded with the expansion strategies. By actually looking at total orders, the simulation procedures were potentially able to make very accurate predictions concerning the potential value of a given expansion. In order to avoid the possibility of doing an exponential amount of work in performing its evaluations, the simulation routines employed a strong heuristic component (for details see [12]). The hope was that we could come up with a theory that would allow us to apply the simulation routines to a subset of those tasks contained in the temporal database. Unfortunately, we never developed such a theory and FORBIN was forced to apply the heuristic task scheduler, as the simulation routines were collectively called, to the entire set of tasks. This oversight on our part prevented FORBIN from solving any really complicated tasks. FORBIN relied upon the heuristic task scheduler for performing the detailed reasoning required for handling travel time and continuously changing quantities. It is our opinion that the knowledge necessary for restricting the application of the heuristic task scheduler would be largely domain specific, as our attempts to derive a purely domain-independent theory met with little success.

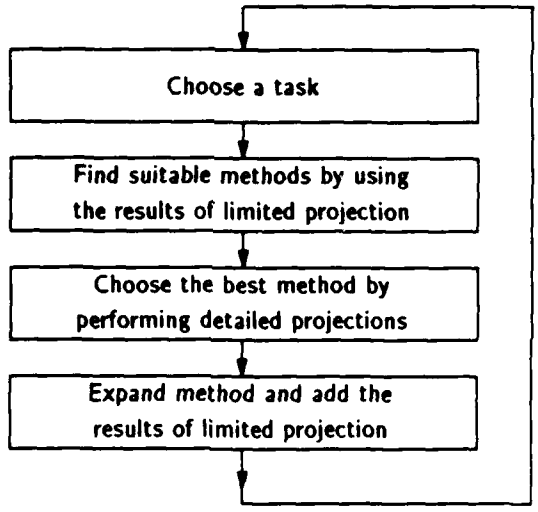


Figure 4: FORBIN algorithm for task expansion

Ideally, the task expander would begin by consulting the entire temporal database

to determine a small set of possible expansion strategies. The algorithm would then use the heuristic task scheduler to score the possible strategies by considering a set of tasks considerably smaller than the entire temporal database. Finally, the task expander would choose that strategy with the highest score and incorporate it into the temporal database. The basic cycle of activity is sketched in Figure 4. It is this method for exploring total orders and scoring expansions using domain-specific utility functions that enables FORBIN to construct "good" plans and avoid backtracking in many situations.

3.5 The Basic FORBIN Architecture

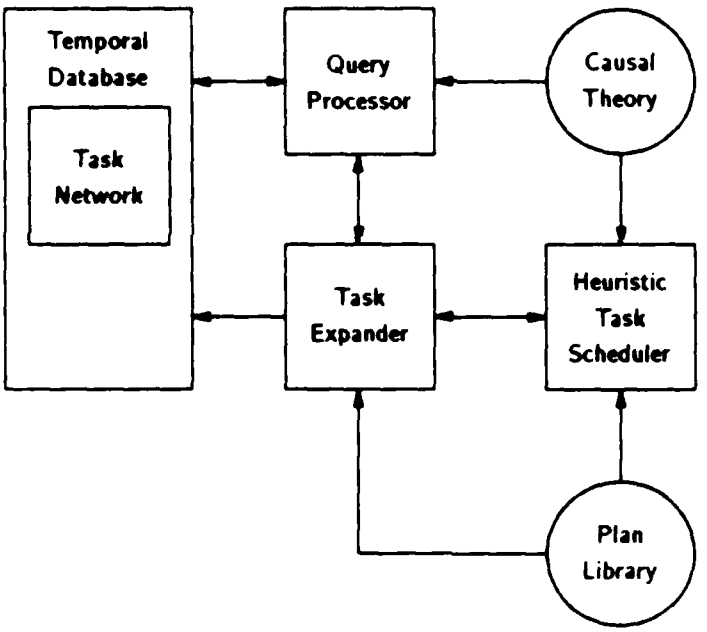


Figure 5: FORBIN architecture

Figure 5 shows the architecture of the FORBIN planner simplified in order to emphasize what is interesting about FORBIN. The temporal database manager is responsible for selectively caching deductions (limited projection) concerning the state of the partially constructed plan. The task expander implements the algorithm

sketched in Figure 4. The heuristic task scheduler handles the detailed predictions required for selecting among methods. In addition, FORBIN employs a *task queue manager* (not shown in Figure 5) that keeps track of the tasks in the database that still need further refinement and decides on the best order to make those refinements.

An important consideration when designing the FORBIN planning system was that planning and execution might occur together. In particular, refinement should concentrate on those tasks to be executed earliest so that their constituent primitive actions can be "peeled off the front" of the plan and executed even while tasks later in the plan are unexpanded. Also, new tasks should be allowed to be introduced by the user at any time and should be incorporated into the growing and executing plan with as little disruption as possible. The task queue manager handles both of these duties in a uniform manner by giving priority for refinement to those tasks whose deadlines are approaching and by inserting user added tasks into the queue just like subtasks spawned through expansion. The FORBIN goal of concurrent planning and execution has been only partially fulfilled so far, but the need for a system with the right properties has been a driving force behind its structure.

3.6 Summary

The description of the FORBIN planner given above partitions the system into two major subcomponents: the domain knowledge in the causal theory and the expansion library and the planning algorithm as embodied by the task expander, temporal database and the task queue manager. The FORBIN plan language is used to describe only the domain-specific knowledge, or that part of the planning system that can change from problem to problem. The algorithmic part of the system is basically a search mechanism with several built in domain-independent heuristics. Since these heuristics are domain independent, they have been incorporated directly into the code for various parts of the planner and are not represented explicitly in a language of their own.

The next two sections of the paper describe the language used to encode the domain-specific knowledge of the FORBIN system. The domain knowledge appears in two places, the causal theory and the task expansion library. The causal theory represents all that is known about the way the world works. The causal theory is used by the temporal database to figure out what facts will be true in the world at all times in the future. The expansion library, on the other hand, represents all of

the ways that the system knows how to do things. Expansions are used by the task expander to generate detailed plans for accomplishing the high level abstract goals given by the user.

4 The FORBIN Plan Language: Causal Theory

The FORBIN system gives the calculation of expected states of the world a central role in plan construction. Each new task added to the plan is designed to have a particular effect on the world but typically it will have that effect only if the world is in an appropriate state to begin with. For example, putting down a cup will not leave it on the kitchen table unless we are, at an absolute minimum, in the kitchen. To ensure that planned actions achieve their intended effects, the task network must be arranged so that each task is predicted to begin execution in the correct world state. The FORBIN planner makes such arrangements in two ways: first, the initial choice of how to expand a task takes into account the possible futures predicted from the existing task network, and second, FORBIN continually adds constraints to the task network in order to guarantee that existing tasks will have their intended effects. Predicting the possible futures of a task network and checking that it is compatible with all known constraints is called projection.

What sort of knowledge representation is required to make FORBIN projection successful? First, and probably most important, is a realistic notion of time. When the success of a plan depends only upon the relative order of tasks, temporal information consisting of order alone is sufficient to calculate expected future world states. However when tasks have completion deadlines and when facets of the world can vary continuously through time, the duration and starting time for every action in the plan must be known as well. Two tasks that start at the same time may generate very different futures according to which takes longer. Proper projection requires knowledge of the duration of each task in the task network, the starting point of every change to the world state caused by a task, and the way that each change of state propagates forward in time.

The second factor that influences the success of FORBIN projection is the expressiveness of its state representation or *causal theory*. In keeping with a realistic underlying representation of time, the causal theory must contain enough information to specify not just the effect each task in the plan has on the world, but also how those effects evolve through time once the task is complete. In most previous planning systems, causal theories are composed of rules indexed by action and specifying a list of assertions to add and a list of assertions to delete in order to model the action being executed. The world state at any one time is seen as a set of assertions like `(ON A B)` or `(COLOR X RED)`. By applying the add and delete lists

from each action in the plan in turn, the expected world state at all times can be determined. Projection rules employing simple add and delete lists are quite powerful even in complex worlds but they fall short of dealing with actions that have context dependent effects or actions that begin processes that change continuously on their own. For example, the action (DEPOSIT $10) might have the effect of increasing one's bank account by $10. No predetermined assertion can be made that reflects the new account balance; it must be calculated from its previous value. Similarly, the action (TURN-ON FAUCET) might start filling the sink with water but no set of add and delete lists alone can predict when it will be filled. A planner must be able to represent both the *absolute effects* handled by add and delete lists and the *relative effects* whose changes to the world state depend on context.

4.1 Specifying the Causal Theory

The FORBIN plan language requires that all absolute and relative effect types be declared before planning begins. These declarations specify every facet of the world that will be needed for projection and how each facet will evolve through time. These declarations along with descriptions of how each type of task is expected to effect the world make up FORBIN's causal theory.

The causal theory comes in two parts: *world physics* declarations and *task descriptors*. The world physics contains the system's knowledge about how every facet of the world should be modeled for projection. For example, FORBIN world physics states that there are such things as lathes, that they can only be used to make one thing at a time, that if one is turned off it will stay off unless acted upon explicitly and so on. The world physics mentions all fact types to be used in projecting the future, how these facts evolve through time once they become true and how different facts interact with one another if they become true at the same time.

A task descriptor encodes the way that a task is expected to change the world when it is executed. Task descriptors are analogous to the add and delete lists of past planners but extend the idea to include statements about relative facts as well. For example, the description of a (RUN-LATHE ...)[2] task might specify that a metal

[2]In the FORBIN plan language, all tasks, facts and states are referred to in terms of *patterns*. A pattern is a list with the first element representing the task, fact or state type. Remaining elements in the pattern are arguments that modify the type. Arguments will often appear as variables in the language but during planning all variables will be instantiated.

blank will be used up, some volume of shavings will be produced and a new product will be brought into being. Task descriptors are the means by which knowledge of the world physics is connected into the task network. All three things, the world physics, the task descriptors and the task network, are required for projection of the expected future.

4.2 World Physics

The world physics section of the FORBIN causal theory describes all facets of the world state that will be modeled. There are two subsections to the world physics: one to deal with absolute fact types and the other to deal with relative fact types. Both types of fact are used in task descriptors to define task effects. Absolute fact types are specified with the exact new state the task puts them into and relative fact types are specified with the change that the task causes in their value. For example, the task descriptor for `(MOVE-TO WIDGET IN-BOX)` might specify that the absolute fact `(LOCATION WIDGET IN-BOX)` becomes true while the relative fact representing the weight of the box will be increased; `(BOX-WEIGHT +(WEIGHT-OF WIDGET))`. More generally, when a task descriptor gives the new state of an absolute fact, that state will not depend on context and will persist until explicitly changed by another task. Conversely, a task descriptor specifies a change to a relative fact and the new state of the fact will depend on its value beforehand and once set in motion, that value may continue the change without further intervention.

The distinction between absolute and relative effect types is not logically required since absolute effects are just a subset of relative ones. However, the FORBIN system separates the two for pragmatic reasons because different processing is done with each during projection. The results of projecting absolute effect types make up the temporal database and place constraints on the starting times of tasks in the task network. The results of the more detailed projection of relative fact types are used to ensure that new additions to the task network are compatible with commitments already made and are not cached in the temporal database.

4.2.1 Absolute Fact Types

Absolute fact types represent facets of the world state that are made true and remain true until explicitly changed. In the FORBIN system they are modeled as predicate

calculus assertions and defined with the form:

```
(DECLARE-FACT (<name> <var1> <var2> ...))
```

For example, many of the properties of a lathe are absolute in nature and might be defined as:[3]

```
(DECLARE-FACT (SETUP-LATHE ?lathe ?type))
(DECLARE-FACT (BIT ?lathe ?type))
(DECLARE-FACT (STATUS ?machine ?status))
```

All static features of the world that are to be referred to by task descriptors must be declared as absolute fact types in this way.

Once an absolute fact type has been defined, the projection machinery knows that each time such a fact is generated by a task, it will remain true until specifically altered. The traditional way of altering the truth of an absolute fact is to delete it and then add a new fact reflecting the new state. However, the explicit deletion of a fact is unnecessary in the FORBIN system because of clipping rules. Clipping rules state that the beginning of one fact causes the end of another. For example, in the machine shop asserting `(STATUS LATHE-1 RUNNING)` causes the fact `(STATUS LATHE-1 IDLE)` to stop being true. There is no need to explicitly delete `(STATUS LATHE-1 IDLE)` as long as this rule is known. Clipping rules are specified with the form:

```
(DECLARE-CLIP <fact 1> <fact 2> <condition fact>)
```

which states that whenever **<fact 1>** becomes true while the **<condition fact>** is true, it causes **<fact 2>** to end (and it vice versa). For the example above one might use:

```
(DECLARE-CLIP (STATUS ?machine ?status1)
              (STATUS ?machine ?status2)
              (NOT (= ?status1 ?status2)))
```

[3]Variables are specified by prefacing their name with a '?'. This notation is drawn from the DUCK programming language [11] used to implement much of the FORBIN system.

With clipping rules, task descriptors only have to assert those absolute facts they make true. Those facts that they clip are taken care of automatically.

The FORBIN causal theory can also deal with interactions between absolute fact types. The mechanism for interaction is the temporal forward chaining rule and several types of rule are available. The two most important rule types are **DECLARE-OVERLAP** and **DECLARE-CAUSE**. Both types of rule state that whenever a certain set of absolute facts are found to overlap in time, a new fact should be asserted as true. The difference is that with **DECLARE-OVERLAP** the new fact extends exactly over the interval of time that the initiating facts overlap while the new fact spawned by **DECLARE-CLAUSE** persists indefinitely (unless clipped) once has begun.[4] The basic temporal forward chaining rule syntax is:

```
(DECLARE-[OVERLAP | CAUSE] (AND <fact 1> <fact 2> ...)
                                <result fact>)
```

For example, to state that whenever the lathe and milling machine are running at the same time it will be very noisy one might write:

```
(DECLARE-OVERLAP (AND (STATUS LATHE RUNNING)
                      (STATUS MILLING-MACHINE RUNNING))
                 (NOISE-LEVEL HIGH))
```

while stating that the same overlap of states will crack the window takes the form:

```
(DECLARE-CAUSE (AND (STATUS LATHE RUNNING)
                    (STATUS MILLING-MACHINE RUNNING))
               (HEALTH WINDOW CRACKED))
```

A final special case of absolute fact t[illegible]s the *pool*. A pool is a collection of objects that are essentially interchangeable and which are referred to throughout the plan language primarily by type rather than name. By referring to a pool, an object of the appropriate type can be identified. Pools are managed by an interconnected set of facts and clipping rules but a simpler syntax is adopted by FORBIN because of their frequent use. A pool is initialized with the form:

```
(DECLARE-POOL <type> (<member 1> <member 2> ...))
```

[4]The Temporal Database Manager which manages the interaction of absolute facts actually supports several more types of temporal chaining rules. However, rules other than these two were used infrequently in our trials. For further discussion of temporal reasoning and the abilities of the TDM see [5].

For example, a machine shop containing two identical lathes might have them represented as a pool from which either can be selected for a particular job:

```
(DECLARE-POOL LATHE (LATHE-1 LATHE-2))
```

The plan expansion language contains forms for reserving an object in a pool and returning it when finished. Pools are used extensively throughout FORBIN example problems to manage discrete resources.

Absolute fact types, clipping rules and temporal forward chaining rules give the FORBIN planner a powerful way to model many facets of the world (see [5] for more background on reasoning about absolute fact types). However, absolute fact types are only good for describing task changes that can be determined before much of the task context is known. To go further, we must introduce relative fact types as well.

4.2.2 Relative Fact Types

Relative facts are used to represent those facets of the world that can best be modeled as a single continuous quantity.[5] Absolute facts change value instantaneously and are always specified in terms of their new value. Relative facts on the other hand are often specified in terms of some change to their value (and hence the new value depends on the old) and once set in motion a relative fact may continue to change over time without further intervention. In the FORBIN plan language, relative facts are called *states* and are referred to by way of a pattern similar to those for absolute facts and tasks; with the first symbol in the pattern is the name of the fact type (or state) and the second is the new value or change. During projection, new values for states will be asserted by tasks in terms of such patterns.

The schema for a state is shown below and consists of four parts: a name and three functions called MOVE, DELAY and UPDATE.

[5]The syntax given here for states is a simplified version of what is possible. A state can be generalized from a scalar quantity to an arbitrary LISP object and can carry a great deal of internal information around. The three state functions can then refer to this information in an arbitrary manner. For such examples the patterns used to refer to the state in task expansions and descriptor will have more arguments. See [12] for more background on the Heuristic Task Scheduler used to reason about state representations.

```
(DECLARE-STATE <name>
   (MOVE   (lambda (<current-value> <change> <time>) ...))
   (DELAY  (lambda (<current-value> <change> <time>) ...))
   (UPDATE (lambda (<current-value> <change> <time>) ...)))
```

Each of the functions takes three arguments, the current value of the state, the change desired in the state value and the current time. During the projection, these functions are called by the system to find out whether, and then how, a state can be changed as specified by various tasks. The current value and time are filled in by the system for the temporal interval of interest and the change value is filled in with what appears in the task descriptor for the task being examined. The MOVE function returns a boolean value that says whether the desired change is possible and the DELAY function returns an estimation of the time that it will take to carry that change out. The UPDATE function simply returns the new value of the state after the change has been made. For example, the state for a robot's location might be called POSITION with a current value of AT-LATHE-CHUCK. If a task has the effect of changing the robot's location to AT-WORKBENCH then the MOVE function applied to AT-LATHE-CHUCK, AT-WORKBENCH and the current time would return whether or not the move from one to the other is possible, the DELAY function would return how long it will take and the UPDATE function would return the new value of the POSITION to be AT-WORKBENCH.

The three state functions are written by the causal theory builder to reflect the various properties of different relative fact types. In the robot position example above, the changes to POSITION that the plan language will specify are new locations for the robot to be. However, it is just as easy to write state functions so that the changes specified in the plan language are relative to the current value. For example, the robot might have a store of plastic resin that is often taken from and occasionally replenished. Such a state could be represented as shown in Figure 6.a. Notice that in the PLASTIC-RESIN state functions, the change is treated in a relative way. In the plan language such changes would be represented as (PLASTIC-RESIN +10) or (PLASTIC-RESIN -10).

An important final aspect of state definitions is that the UPDATE function may generate action requests of the form: (ACTION <pattern>). The patterns in action requests correspond to primitive tasks and become incorporated as part of the final plan that FORBIN builds. During the planning process itself, however, action re-

Example A.

```
(DECLARE-STATE PLASTIC-RESIN
   (MOVE   (lambda (old change time) (positive? (+ change old))))
   (DELAY  (lambda (old change time) 0))
   (UPDATE (lambda (old change time) (+ change old))))
```

Example B.

```
(DECLARE-STATE POSITION
   (MOVE   (lambda (begin end time)
             (and (member? begin *valid-locations*)
                  (member? end *valid-locations*)
                  (know-route-between? begin end))))
   (DELAY  (lambda (begin end time)
             (/ (distance-between begin end) *nominal-speed*)))
   (UPDATE (lambda (begin end time)
             (ACTION (move-primitive begin end))
             end)))
```

Figure 6: Examples of State Declarations

quests are essentially ignored and play no part in decisions that influence the plan. The reason for the existence of actions requests is to allow the task network to grow without regard to certain necessary tasks and yet have those tasks added into the network at the finish. The most common example of this is robot position. In the FORBIN domain, the robot must travel around from work-station to work-station and the final plan must contain a primitive task for each change of locale. However, the start and end position of each travel task cannot be determined until all of the other parts of the plan have been ironed out and the planner knows where the robot needs to be for every task. Thus, once the final task network has been determined, a last run through it is made with the projection machinery; any action requests produced are added to the plan. At all other times action requests become NO-OP's. For example, the position state for the robot might be encoded as shown

in Figure 6.b. This ability to add actions as required at projection time is used extensively in FORBIN example problems to deal flexibly but effectively with travel time.

4.3 Task Descriptors

The world physics describes the way that facets of the world state are expected to change through time but it does not describe exactly how those facets are expected to be changed by particular actions of the planner; the FORBIN causal theory captures this information in *task descriptors.* Each type of task in the system has a task descriptor that specifies every fact that will be altered by that task and the way that it will change. The task descriptor also gives the expected duration for its task. An important aspect of FORBIN task descriptors is that they specify only the *expected* behavior of a task because there may be several different ways for the task's expansion to be carried out. For example, given the task of making a widget it may be possible to use either the drill press or the milling machine. Each of these expansions will tie up a different machine and will alter a number of different facets of the world. However, the (MAKE WIDGET) task must be incorporated into the task network before an expansion can be chosen and its expected behavior must be included. This behavior will be specified in the task descriptor for the (MAKE WIDGET) task which must make some compromise between the facts that might be changed by the drill press expansion and the facts that might be changed by the milling machine expansion. In general, task descriptions give the expected behavior of a task before knowing how the task will actually be carried out and thus can only be an estimation of the tasks actual effects.

```
(TASK-DESCRIPTOR <id>
   (TASK <pattern>)
   (EXPECTED-DURATION <low> <high>)
   (EXPECTED-STATES ...)
   (EXPECTED-FACTS ...) )
```

Figure 7: The Basic Form of a Task Descriptor

The schema for a task descriptor is given in Figure 7 and consists of four sections: The TASK section of the descriptor identifies the task type to which this descriptor applies, the EXPECTED-DURATION gives an estimation of how long this type of task will take. The EXPECTED-STATES and EXPECTED-FACTS sections together specify the effects that this type of task is expected to have on the world, when executed.

4.3.1 Task Specifier

The task specification section of the descriptor takes the form of a list to be matched against the task specifications given in the expansion methods described in the next section. Whenever a task is added to the FORBIN task network, a task descriptor with matching specification is taken from the causal theory and used to describe the effects of that task to the projection machinery. Task descriptor specifications may contain variables and hence it is possible for a task in the network to match more than one descriptor. For this reason task descriptors are ordered (like prolog clauses) and the first one to match a task is chosen. Tasks in the task network will never have variables in their specifications so, after matching, all of the variables in a task descriptor will be instantiated. Some example task specifications are:

```
(TASK (MAKE ?thing))
(TASK (SETUP-LATHE ?lathe WIDGET))
(TASK (SETUP-LATHE ?lathe ?type))
```

4.3.2 Expected Duration

The expected duration of each task in the task network is crucial when temporal considerations are important to the success of a plan. Planning for deadlines requires knowledge of how long execution of all tasks in the plan is expected to take. This information is specified in the expected duration section of the task descriptor for each type of task. A duration is given as an estimated interval with a lower and upper bound. During projection these intervals are used to decide which facts will be true when. Some sample duration specifications are:

```
(EXPECTED-DURATION 5.0 6.0)
(EXPECTED-DURATION (* 3 (SIZE-OF ?thing)) (* 10 (SIZE-OF ?thing)))
```

Once an expansion is chosen for a task, its expected duration is replaced with the composite durations of the subtasks in the expansion. Thus, as tasks are refined, so are the estimates of how long they will take.

4.3.3 Expected Facts

The expected facts section of the task descriptor specifies those absolute effects that are expected to be made true by the execution of this type of task. The specification is a simple list of facts like a traditional add list. It is assumed during projection that each of these facts will be made true at sometime during the execution of the task. No delete list is required because of the clipping rules specified in the world physics. Each fact is specified as a pattern and must have been previously declared in the world physics section of the causal theory. For example, the following expected facts specification declares that executing this type of task will cause the lathe to be ready, the lathe bit to be of the correct type, and for the robot to be positioned at the lathe's chuck:

```
(EXPECTED-FACTS
   (READY ?lathe ?type)
   (BIT ?lathe (BIT ?type))
   (LOCATION ROBOT (LOC-OF CHUCK ?lathe)))
```

4.3.4 Expected States

The final section of a task descriptor specifies expected states or relative effects. Expected states are given as a list of patterns just like expected facts, but each is augmented with an indication of where during the task the state will take on (or be changed by) the value specified. An expected fact is assumed to become true sometime during the task while an expected state may be specified to change at the start, end, or sometime during the execution of the task. Thus an expected state has the form (<when> <pattern>) where <when> can be one of: START, END or DURING. It is most enlightening to think of expected states as specifying "island states" or places in time when particular changes to relative effects will take place. For example, the following specification for the expected states of a (MOVE ?thing ?loc1 ?loc2) task will put the robot's position at ?loc1 to start and ?loc2 at the end:

```
(EXPECTED-STATES
   (START (ROBOT-POSITION ?loc1))
   (END (ROBOT-POSITION ?loc2)))
```

As with the expected duration, after a task is expanded, its expected facts and states are replaced with those of its composite subtasks.

4.3.5 An Example

To pull everything together, consider the complete task descriptor in Figure 8. This descriptor specifies all of the causal information required for projection of the effects of lathe setup tasks. It says that any lathe setup will be expected to take from 5.0 to 6.0 time units and will result in the lathe being set up for the correct type of operation with the correct bit installed. Also, at the end of the task the robot will be positioned at the lathe's chuck. Whenever a task of this type is added to the task network, it will be annotated with all of this information and from then on the projector will know how that task will change the world upon execution.

```
(TASK-DESCRIPTOR SETUP-T1
   (TASK (SETUP-LATHE ?lathe ?type))
   (EXPECTED-DURATION 5.0 6.0)
   (EXPECTED-FACTS
      (SETUP-FOR ?lathe ?type)
      (BIT ?lathe (BIT ?type))
      (LOCATION ROBOT (LOC-OF CHUCK ?lathe)))
   (EXPECTED-STATES
      (END (POSITION (LOC-OF CHUCK ?lathe)))) )
```

Figure 8: The Task Descriptor for a Lathe Setup Task

This task descriptor also illustrates an interesting point with respect to the different processing of facts and states. Notice that there is both an expected fact entry for the robot's `location` and an expected state entry for the robot's `position`; obviously these two entries represent the same facet of the world and will take on the same values. However, the `location` fact models the robot's location as an absolute fact that can be changed at will while the `position` state models it as a continuous quantity that requires action requests to change it. The results of projecting the two together should be the same as projecting either one, except that the `position` state

will have more temporal accuracy. The reason for using both is that the results of projecting **location** facts will be cached and can be used for later temporal queries while the **position** states will only preserve overall plan consistency. On the other hand, the **position** state will add the action requests necessary to put actual robot movement tasks into the final plan. Thus, the two complement each other with the **location** used as a basis for flexible future plan choices and the **position** used to add detail during consistency checking.

4.4 Summary

In summary, the causal theory is used by the FORBIN planner to search through the task network being built and project the expected world state at any point in time covered by the network. This projected future is used to ensure consistency in the task network so far (*i.e.* the states required by all tasks in the network will be true as expected) and as a basis for the choice of appropriate, detailed tasks to add to that network. Projection requires that each task in the task network be tagged with an expected duration and those facets of the world that the task is expected to change. This information is contained in FORBIN task descriptors which specify the expected behavior of each type of task even before a method for carrying out that task is chosen. In addition to task descriptors, projection requires that the evolution and interaction of the facts be specified for each task as well. That way the values of each type of fact can be calculated between tasks even if they change without being specifically acted upon. This information is captured in the FORBIN world physics with its facts, states, clipping rules and temporal chaining rules.

Within the FORBIN system projection is carried out in two phases with the results of one cached for efficiency and the results of the other dropped to maintain flexibility. The first phase of projection is the maintenance of a temporal database based on absolute facts, clipping rules and temporal forward chaining rules. This database contains those absolute facts that can be determined to be true given the partial ordering of tasks in the current task network. In general many different futures are possible for a given task network corresponding to different total orderings of the tasks. The database caches only those facts that are unambiguously true regardless of possible orderings (with respect to absolute facts) and is thus incomplete in terms of what will be true in detail in the future. Despite its incompleteness, this database is used by FORBIN as its basis for further planning decisions.

The second phase of projection involves the exploration of total orderings of the task network and the resulting evolution of any relative facts. Since these facets of the world often have values that depend on what state they were in before they were changed they can only be projected when the order of all tasks have been determined and each has a known predecessor. As long as a total ordering of the task network can be found that allows each absolute and relative fact type to take on the values desired of it at the correct time, then the plan developed so far is still consistent and workable. In general the cached futures determined by projecting only absolute fact types may seem consistent until the added information contained in the relative fact types is included too. This check for detailed consistency is made whenever a new task is to be added to the task network. Once the network has been deemed consistent the ordering discovered in checking it is discarded because there may be many such orders and the one discovered may not work once more task are added to the network. These consistency projections are used in deciding whether a proposed task expansion is reasonable or not.

In the FORBIN system projection is an active process. Many tasks in the task network will carry assumptions, or expected world states, that must be satisfied when the task is executed. To ensure that this is done, each task in the task network is allowed to "float" in time and can be positioned by the projection machinery to start when its assumptions are met. This floating is constrained by the orderings and deadlines imposed in the task network. As new commitments are made and new tasks added to the network, tasks already there may be moved with respect to each other to keep their assumptions satisfied. Thus, FORBIN projection not only keeps track of what is expected in the future as a basis for task expansion, it also keeps task assumptions true by positioning each task in time. A central feature of the task expansion algorithm described in Section 6 is that it tries very hard to never make a commitment that would make it impossible to meet the assumptions of an existing task. Should such a situation arise, the task network would no longer represent a consistent, viable plan.

The causal theory and the projection mechanisms in the FORBIN planner provide a powerful means for predicting the future state of facts corresponding to the consequences of tasks in the task network. These facts are used as a basis for expanding the task network into more detail and thus building a plan. The next chapter of the paper introduces the representation for task expansions and how they are used to grow the network based on projected futures.

5 The FORBIN Plan Language: Expansion Library

The plan expansion library contains the methods that FORBIN has available to it for accomplishing its tasks. The system cannot build a plan for a task that does not have an explicit expansion in the library. Expansions can be parameterized however, so that an expansion might apply to many tasks of the same type. For example, in the simple machine shop domain used by FORBIN one of the high level task types is `(MAKE ?thing)`. Here, `?thing` is a parameter meaning separate methods are not required for `(MAKE WIDGET)`, `(MAKE GIZMO)` and so forth. This requirement of known expansions for all task types is a general feature of hierarchical planners because the hierarchy (embodied in the expansion strategies) must be defined in advance. Earlier planners like STRIPS [8] were more general in that they derived a plan from the causal theory by backward chaining on the effects defined by task descriptors. These chaining planners did not need a plan library and could build a plan to achieve any effect mentioned in their causal theory. On the other hand, chaining planners have a much larger search space to explore because of the enormous (possibly infinite) number of chains that can be built that don't lead to a final solution.

The expansion hierarchy used by the FORBIN system consists of a library of *task expansions*. Each entry in the library describes one way of carrying out an abstract task. When there is more than one way to carry out a task, then there is more than one entry in the library. For example, it might be possible to build a new gizmo with either the lathe or the milling machine. Since each one requires a different set of bits and setup procedures, it would make sense to enter each one into the library as a separate expansion. Each expansion entry is known as a *method* for expanding its type of task.

This section of the paper describes the notation used to specify task expansion methods.

5.1 The Expansion Method

A FORBIN task expansion method presents one possible way to carry out a task. There may be many ways to accomplish a particular task in a given domain so there will often be several expansion methods for the same task in the FORBIN library. Each method consists of four sections as shown in Figure 9. The TASK section of the method is used as an index and specifies the type of task that the method can be

used to accomplish. The **ASSUMPTIONS** section describes characteristics that must be true of the world state before and during execution of this method to ensure that it will perform the desired task correctly. The actual subtasks, and their order, are detailed in the **SUBTASKS** section of the method, and the **UTILITY** section gives a way of comparing this method with other methods that perform the same task. All other things being equal, the highest utility method with satisfied assumptions is the best one to choose.

```
(DEFINE-METHOD <identifier>
   (TASK <pattern>)
   (UTILITY <number>)
   (ASSUMPTIONS <assumption1>
                <assumption2> ...)
   (SUBTASKS (<tag> <pattern>
                    <reason1> FOR <tag>
                    <reason2> FOR <tag> ...)
             (<tag> <pattern>
                    <reason1> FOR <tag>
                    <reason2> FOR <tag>) ...))
```

Figure 9: The Basic Form of a Task Expansion Method

The **<identifier>** of the plan descriptor is a unique symbol used to differentiate one expansion method from another. It plays no role in the FORBIN algorithm. The other elements are described in some detail below.

5.1.1 Task Type

The (**TASK <pattern>**) portion of the plan desciptor is used to identify what task this method is an expansion for. The **<pattern>** portion may contain parameters or constants depending on whether the method may be used as an expansion for several tasks or just one. The **TASK** field serves an identical function in expansion methods as it does in the task descriptors described in the previous section.

5.1.2 Utility

The (UTILITY <number>) portion of the plan descriptor is used to give relative "goodness" ratings to the various methods of carrying out the task. When a task has more than one plan descriptor, the one with the highest utility is given some small degree of preference over the others by the Heuristic Task Scheduler (HTS) when it is choosing the best plan to use.

The utility value can either be a constant or a lambda expression. Thus the utility for a particular expansion can vary depending on the exact task for which the expansion is to be used. For example, consider:

```
(UTILITY (lambda () (cond ((= ?type widget) 4)
                          ((= ?type gizmo) 2))))
```

The method containing this statement has a higher utility if it is operating on widgets than if it is being used for working on a gizmo.

5.1.3 Assumptions

The (ASSUMPTIONS ...) section in the plan descriptor details those things that must be true before the plan can be used as a method for completing the task. Each assumption is a predicate that must be true in the temporal database. All of the assumptions together make up a conjunctive query to the database. A valid assumption is any predicate that might be true in the temporal database, but for the most part, the only predicates necessary are:

- (TT <begin> <end> <effect>) stating that the formulae <effect> must be true throughout the time interval <begin> to <end>.
- (RESERVE <begin> <end> <pool> <thing> <tag>) stating that the object <thing> from the pool <pool> must be available to be reserved by the task <tag> throughout the time interval <begin> to <end>.

Throughout an expansion method, the special symbol *SELF* stands for the task that is being expanded. For example, the assumptions necessary for the GENERATE task in the FORBIN domain are:

```
(ASSUMPTIONS
   (RESERVE (BEGIN *SELF*) (END *SELF*) LATHE ?LATHE (TAG *SELF*)))
```

The assumption states that this method will only work if a LATHE can be reserved for this task over the entire duration of the task. LATHE refers to a pool of lathes; several machines, any of which would be suitable for performing the GENERATE task. ?LATHE will be bound to a particular machine in the lathe pool. For a more detailed discussion of the RESERVE predicate see [5].

The MOVE method of Appendix B requires the TT assumption:

```
(ASSUMPTIONS
   (TT (BEGIN *SELF*) (BEGIN *SELF*) (LOCATION ?THING ?START)))
```

This assumption demands that the fact (LOCATION ?THING ?START) be true during the interval defined by the start of that MOVE expansion method. In other words, that fact must be true at the start of the move task if this method of doing the move is to be successful.

In most cases where there are alternative methods for accomplishing a task, the assumptions will play an important role in choosing a particular method. The assumptions for a particular method constrain the time and order in which that particular method may be used in the overall plan. The HTS, using the assumptions, picks the method that fits in "best" with the overall plan, as already developed. Thus, different assumptions will cause different methods to be picked — causing an overall different plan to be created.

5.1.4 Subtasks

The (SUBTASK ...) portion of the expansion method is used to define the piece of the task network into which the task should expand. This portion of the method includes some number of subtasks; each identified within the method by a <tag>. Following the tag is some <pattern> which should match the TASK pattern of one or more expansion methods in the task library. After each pattern is a list of <reason>s why that subtask should be done. Reasons often will reference the tags, thereby defining some partial order over the subtasks.

There are three general types of reasons for a subtask:

- (ACHIEVE <effect>): the purpose of the subtask is to create this effect. The effect will be entered into the temporal database.

- (CREATE <pool> <object>): this subtask creates a new object <object> that can be added to the group of objects specified by <pool>.
- (CONSUME <pool> <object>): the opposite of CREATE, this reason specifies an object to be taken out of a pool.

The reasons for a subtask are connected to other subtasks with the FOR operator. FOR indicates that the reason given is to satisfy a precondition for the execution of the named subtask (referenced by its tag). These reasons, and the subtasks that they are for are then loaded into the temporal database where the projection machinery forms the appropriate partial order. Normally, all of the subtasks are constrained to fall within the temporal bounds of the task being expanded (*i.e.* *SELF*). However, the FOR operator has three other forms: <FOR, FOR>, and <FOR>; which indicate that the indicated subtask may be positioned before, after, or on either side of the main task for which it is in service.[6] For example, the INSTALL-1 method has the two subtasks shown in Figure 10. The first subtask takes the old bit out of the lathe chuck so that the lathe chuck will be empty. By using the <FOR, the fact that the lathe chuck is empty as a precondition for doing the second subtask is added into the temporal database. The < indicates that the first subtask may be accomplished any time prior to the second subtask, so long as the fact that the lathe chuck is empty remains true till the second subtask is started.

```
(SUBTASKS
   (T1 (REMOVE-BIT ?OTHER-BIT ?LATHE)
       (ACHIEVE (BIT ?LATHE NONE)) <FOR T2)
   (T2 (PUT ?BIT (LOC-OF CHUCK ?LATHE))
       (ACHIEVE (BIT ?LATHE ?BIT)))))
```

Figure 10: The Subtasks for the INSTALL-1 Method

The REMOVE-BIT method has a slightly different subtask scheme (see Figure 11). The first subtask must be performed before the second and during the scope of

[6]The FORs may have associated with them metric intervals that specify exactly how close or far two subtasks may be separated in time. These intervals have been left out of this paper to help maintain clarity.

the REMOVE-BIT task (as shown by the vanilla FOR). The second subtask may be done any time after the first. The FOR> indicates it is not bound by the end of the REMOVE-BIT task.

```
(SUBTASKS
   (T1 (GET (BIT ?TYPE) (LOC-OF CHUCK ?LATHE))
       (ACHIEVE (HAVE *ME* ?BIT)) FOR T2)
   (T2 (PUT (BIT ?TYPE) (LOC-OF BIT-RACK ?TYPE)) FOR>)))
```

Figure 11: The Subtasks for the REMOVE-BIT Method

5.2 An Example Method

Now its time to put everything together and show off a full expansion method. Figure 12 shows the expansion methods for the MAKE, GENERATE, and MOVE tasks. From the preceeding discussion it should be fairly obvious how these methods work. What may not be obvious is why they were done this way rather than some other seemingly more simple way. The next section goes through an expansion in detail, keeping track of what is being added into the temporal database. The reasons for the detailed syntax should then become more apparent.

At this point, it seems appropriate to make some general comments about the FORBIN plan language, and plan languages in general. Although able to represent a wide variety of planning situations, the FORBIN language has one troubling characteristic: detailed knowledge about other plans in the library is almost always necessary when writing a new expansion method. We started with the hope that separating the plan language from the causal theory would help make it easy to add additional methods for task expansion because the temporal projection machinery would take care of the details. However, FORBIN is intended to be a very flexible planner, weaving steps from several methods into a coherent whole. To keep the result consistent, appropriate labeling of facts and events must be maintained across all methods — and therein lies the difficulty. Ensuring that subtasks have a hold on consistent labels no matter which other subtasks they combine with becomes, at times, a syntactic odyssey.

One should not construe the above comment to imply that the FORBIN plan language is ill-conceived. Often we considered simplifying our domain to avoid some particularly troublesome representation problem but we did not give in and as a result, we developed a very powerful language able to express plans in domains of considerable complexity; it is only working out the details of a full library of plans for a new domain that causes us concern. This problem is even worse in most other planning systems, but it is most apparent with FORBIN because the domain we chose is more complex. Needing to know the details of other expansion methods when constructing new ones is not a fatal flaw for automatic planners, but it is a serious complication for those who contemplate building planners to operate in non-trivial domains.

```
(DEFINE-METHOD MAKE
   (TASK (MAKE ?TYPE))
   (UTILITY  1.0)
   (ASSUMPTIONS nil)
   (SUBTASKS
      (T1 (GENERATE ?TYPE $NEW-THING)
          (CREATE  ?TYPE $NEW-THING) FOR T2)
      (T2 (MOVE $NEW-THING (LOC-OF SHELF ?TYPE))
          (ACHIEVE (LOCATION $NEW-THING (LOC-OF SHELF ?TYPE))))))

(DEFINE-METHOD GENERATE
   (TASK (GENERATE ?TYPE ?THING))
   (UTILITY  1.0)
   (ASSUMPTIONS
    (RESERVE (BEGIN *SELF*) (END *SELF*) LATHE ?LATHE (NAME *SELF*)))
   (SUBTASKS
      (T1 (SETUP-LATHE ?LATHE ?TYPE)
          (ACHIEVE (READY ?LATHE ?TYPE)) FOR T2)
      (T2 (RUN-LATHE ?LATHE ?TYPE)
          (CREATE  ?TYPE ?THING)
          (ACHIEVE (LOCATION ?THING (LOC-OF HOPPER ?LATHE))))))

(DEFINE-METHOD MOVE
   (TASK (MOVE ?THING ?FINISH))
   (UTILITY  1.0)
   (ASSUMPTIONS
    (TT (BEGIN *SELF*) (BEGIN *SELF*) (LOCATION ?THING ?START)))
   (SUBTASKS
      (T1 (GET ?THING ?START)
          (ACHIEVE (HAVE *ME* ?THING)) FOR T2)
      (T2 (PUT ?THING ?FINISH)
          (ACHIEVE (LOCATION ?THING ?FINISH)))))
```

Figure 12: The Methods for MAKE

6 The FORBIN Planning Algorithm

Section 2 of this paper discusses the concept of representing plans as task networks that might be constructed using various search techniques. Section 3 describes the FORBIN system as a planner that builds its task network using a combination of hierarchical expansion to add detail and temporal projection to ensure correctness. Section 4 describes the causal reasoning methods used to project the expected effects of a partially built task network and Section 5 describes the language used to write expansion strategies for specific task types that might appear in the network. This section of the paper gives a detailed description of the hierarchical search algorithm at the heart of the FORBIN planner.

The FORBIN search algorithm is conceptually quite simple. At any point in the search there is a partially elaborated plan represented by the current task network. The object of the search is to expand each task in the network into more and more detail until only primitive tasks remain. The final plan produced by the system is the network of primitive tasks.

Unfortunately this simple search algorithm is computationally much too expensive to use directly so the FORBIN system makes use of many heuristics to limit the number of partial task networks considered and the number of projected futures calculated. The detailed discussion below attempts to make these heuristics explicit and thus clarify exactly which planning problems FORBIN can solve and which it cannot.

6.1 Introducing a New Task into the Task Network

Throughout the remainder of this section we will illustrate the FORBIN algorithm on a single example in the FORBIN domain. The domain and task expansion METHODS are contained in Appendix B for those interested in the details, while in the text, portions of the task network will be displayed to help clarify the discussion. Initially FORBIN is given the task (or more precisely, tasks) of constructing two items in its factory: a gizmo and a widget. The system is given the further constraint that the widget must be completed (*i.e.*, constructed and properly shelved) by time 35. The system is also given an implicit constraint to finish both projects as quickly as is mechanically possible.

Figure 13 shows the initial task network (the state of the temporal database) after FORBIN has been given its task. The vertical line segments indicate the points in time that the planner will actually deal with. Time flows from left to right. The thin lines connect pieces of the network and represent ordering constraints, but not any definite period of time. The labeled horizontal segments indicate time to be spent doing whatever the label indicates; several activities may be going on in parallel. The network illustrated indicates that at some time after time zero the **MAKE Widget** task will be started. That task will continue on for a while but will end by time 35. Meanwhile, some time after time zero (but not necessarily the same "some time" as in the previous sentence) the **MAKE Gizmo** task will be started. This task will also continue on for a while — but its end point is not presently constrained.

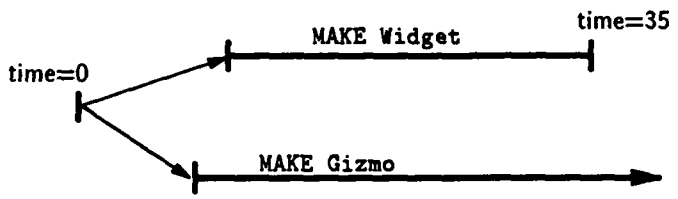


Figure 13: The Initial Task Network for Constructing a Widget and Gizmo

The task network in Figure 13 includes some information from the TASK DESCRIPTORS. The descriptors contain the initial estimates of how long the tasks will take to execute.

Task expansion choices are based on more than the task network shown in the diagram; they are based on the temporal database derived from the task network and the task descriptors. There are many other things actually in the temporal database that are not shown in the picture. It contains everything that is known about the state of the world. The position of the robot, the bit currently in the lathe, and all other tasks known to the system, both past and future, are in the database. However, for simplicity's sake, we will concentrate directly on the tasks at hand and show only the network itself.

6.2 Selecting a Task to Expand

Whenever FORBIN has unexpanded non-primitive tasks in the task network it attempts to expand them. Task expansions are done one at a time and the task to expand is chosen based on several criteria, among them are:

- Since planning takes time, and some tasks have deadlines, tasks with approaching deadlines are given some priority in expansion.
- Tasks that are very high in the abstraction hierarchy should be expanded to sufficient detail that they may be placed in proper perspective in the task network.
- Tasks that are heavily constrained should be expanded so that none of their constraints are accidentally violated.

FORBIN will not retreat on an expansion decision so the order that tasks are expanded may be critical to the plan that eventually evolves. Decisions about how a task should be expanded (*i.e.*, what METHOD to use) are based on the facts in the task network, and those depend on the level of expansion for the tasks in the network. Since FORBIN will not retract an expansion decision the algorithm may fail to find a successful plan where one exists. But the alternative is to commit the system to an exponential, backtracking search that in unfriendly cases may be unable to find a viable plan anyway.

6.3 Queries and Projections of Possible Expansions

Figure 14 shows the task network after both the MAKE `Widget` and MAKE `Gizmo` tasks have gone through one expansion. These expansions are as easy to choose as any performed by other planners like NOAH since there is only one way to expand the MAKE task and it cannot interfere with other tasks in the network. Thus there really is no expansion decision to worry about. Unlike most previous systems, FORBIN's decisions will not always be this easy.

The MAKE plan is very high level. The only details that come out from its expansion are that each MAKE requires two subtasks, the first generates the item, the second involves moving the item to its proper storage place. Both of these subtasks,

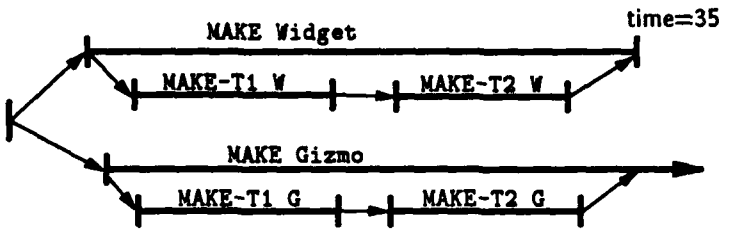


Figure 14: The Task Network After One Round of Expansion

as defined by their FORs, are constrained within the original bounds of their supertask. The unlabeled arrows between the subtasks indicate that the second subtask must follow the first by some amount of time that is only indirectly constrained by the starting constraint on the first subtask and the completion constraint on the second.

The FORBIN algorithm continues cycling, and after a short while the task network looks like that shown in Figure 15. The expansion has concentrated on the WIDGET task because it is constrained by a deadline. The first subtask has been expanded twice to the second's once because after the first expansion of the GENERATE an ASSUMPTION that the lathe is reserved was added to the temporal database. No conflicts with the assumption were detected, so expansion proceeded. The left and right angle brackets linking items in the network (*e.g.*, MAKE-T2 **Widget** and MOVE **Widget**) indicate the passing on of exact constraints (*i.e.*, MOVE **Widget** has exactly the same constraints as the subtask it was expanded from).

After the series of expansions in Figure 15, the balance of choosing a task to expand switches to the GIZMO task. The reason for this switch is that the MAKE **Gizmo** is now at too high an abstraction level with respect to the expanded GENERATE **Widget**.

As the first subtask of the MAKE **Gizmo** is expanded, a query is made using the temporal database machinery. This query asks when the ASSUMPTION about lathe reservation that goes with the GENERATE METHOD will be true. The query returns

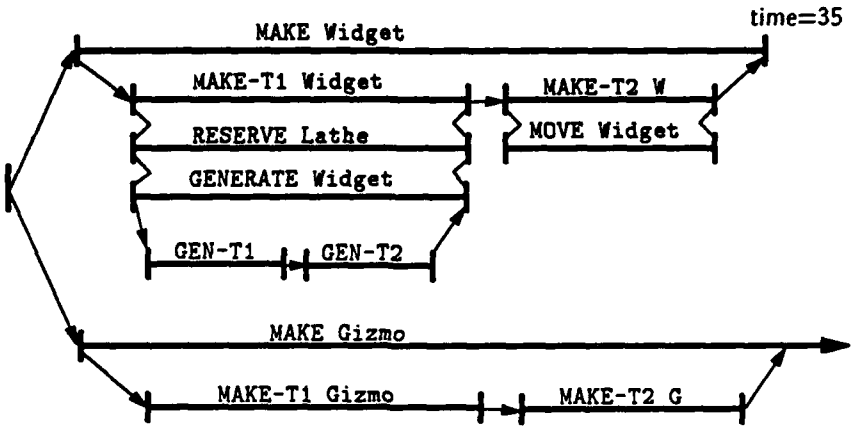


Figure 15: The Further Expansion of the MAKE Widget Task

a disjunctive ordering constraint; the GENERATE Gizmo must occur either before or after the GENERATE Widget. Thus the task network now contains two distinct possible futures: one with the GIZMO made on the lathe before the WIDGET, and an alternative view where the WIDGET is produced first (shown in Figure 16).

It should be noted that both of the orderings may not actually be feasible. First, the tasks have not really been expanded to sufficient detail to know for sure if either will actually lead to a feasible plan. Second, the FORBIN algorithm, during each cycle, runs the task network through more detailed local projection, in order to spot adverse interactions of relative effects. As long as one feasible ordering for the task network can be found, it is left as is.

6.4 Comparing Possible Expansions

So far, in this discussion of the FORBIN algorithm, projection of the task network has been accomplished almost exclusively by the temporal database manager. This machinery has allowed the planner to spot possible trouble spots due to conflicts arising from separate tasks (*e.g.*, the need for a lathe by both MAKE tasks) and it has

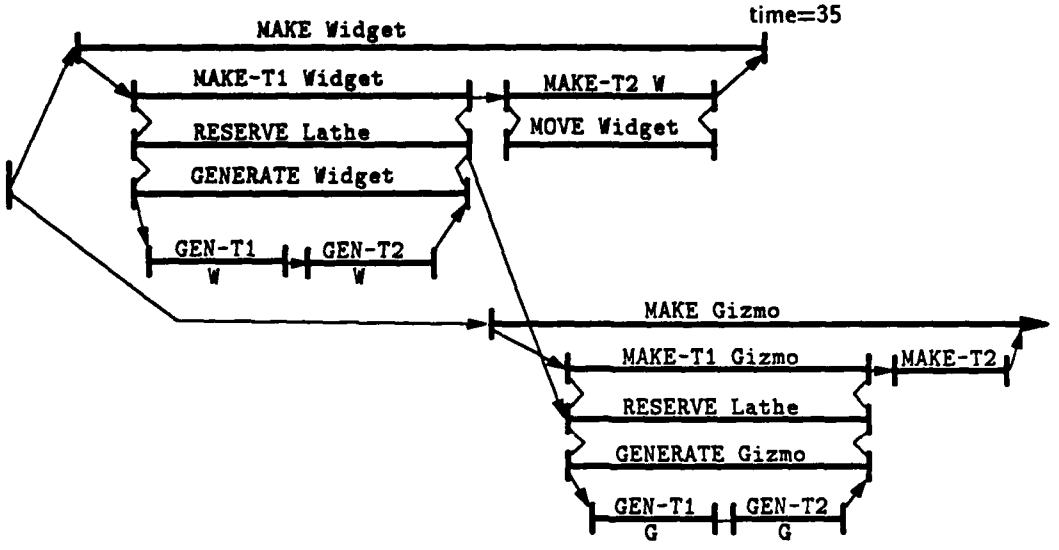


Figure 16: The Further Expansion of the MAKE Gizmo Task

automatically inserted disjunctive constraints to keep these potential conflicts from becoming actual plan bugs. This alone is a more detailed form of plan *critic* than has been previously implemented. However, it is not in itself sufficient when issues of actual plan design appear.

For some tasks FORBIN has several different METHODs for expansion. The first subtask of the GENERATE METHOD is to set-up the lathe. There is one METHOD for expanding this task. It involves the FORBIN robot acquiring the correct lathe bit, and then installing that bit into the lathe. There are multiple METHODs for how to install the bit. This depends on whether there is a bit already in the lathe, and what type of bit this is. Committing to a specific METHOD for installing the bit into the lathe is committing the world to be in a particular state (or at least to having certain facts be true) at the time when that METHOD is to be expanded. A partial order may not be sufficient to guarantee this, and further constraints may have to

be added. The database projection machinery can add the necessary constraints, but it cannot decide on which constraint set to add (*i.e.*, which METHOD to commit to). So when FORBIN gets to the point of expanding the INSTALL task, all options are loaded into the heuristic task scheduler for more detailed projection.

The HTS efficiently explores the space of total orderings, inserting relative effects that will take place because of a specific ordering. As it explores these orderings it may place any of the possible expansion METHODs into the schedule it is building. When the HTS derives a feasible and efficient schedule it notices which expansion METHOD was used in the schedule. This is the METHOD that is finally inserted into the task network.

In the case of expanding the **INSTALL Gizmo bit** task, the ASSUMPTIONS can be met for either of two expansion METHODs. The INSTALL-1 METHOD may be used. The ASSUMPTIONS for this METHOD demand that the lathe already be setup, but for the wrong type of product. Thus this METHOD includes steps for removing the old bit and inserting the new one. These assumptions demand that the **INSTALL Gizmo bit** take place after the lathe has been RESERVED for the **GENERATE Gizmo** task and after the lathe has been set up for the **GENERATE Widget** task.

The INSTALL-2 METHOD can also have its ASSUMPTIONS met. It demands that the lathe contain no bit. This condition also exists in the task network at a time before either **GENERATE** task is done. Using this METHOD causes a schedule to be created where the gizmo is produced first but the HTS projects that this will cause a deadline violation for the **MAKE Widget** task (after all the travel tasks are added in) and the schedule is deemed non-viable. Thus the HTS chooses the INSTALL-1 METHOD for expanding the **INSTALL Gizmo bit** task.

Once this choice is made, the temporal database inserts the necessary constraints to maintain a temporally consistent task network. The resulting network is shown in Figure 17.

It should be noted that the diagram also contains two tasks linked to their supertasks via the **<FOR** operator. The first subtask in the **SETUP-LATHE Gizmo** METHOD has such a link. That subtask involves having the robot acquire the proper lathe bit. That aquisition may be done at any time prior to the installation of the bit into the lathe. The first subtask in the INSTALL-1 method is similarly linked but that task (which involves removing the old bit from the lathe) is constrained by having the lathe reserved.

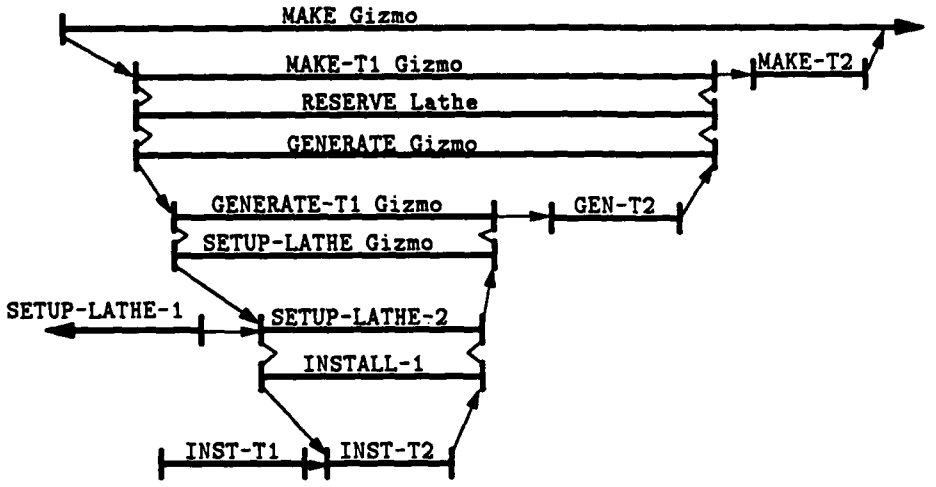


Figure 17: Choosing the INSTALL-1 METHOD

6.5 Summary of FORBIN Algorithm and Solution

The FORBIN algorithm involves a cycle of operations that are performed on the task network. These are:

1. Select a non-primitive task DESCRIPTOR from the network to expand. This selection is based on several heuristics involving the constraints on the tasks and the state of the network. Here the network acts as task queue.

2. From the library of expansion methods, extract all those whose pattern field matches the task being expanded.

3. Using the temporal database projection machinery, query the task network with the ASSUMPTIONS from each of the selected expansion METHODS. Eliminate those METHODS whose ASSUMPTIONS cannot be met. If all methods are eliminated then FORBIN has failed to successfully find a plan for that set of tasks.

4. Have the HTS try and create a viable schedule from the task network and its choice of expansion METHOD. If the HTS is unable to find a viable schedule, FORBIN fails in planning for that set of tasks.

5. If the HTS finds a viable schedule insert its choice of METHOD into the task network.

6. Insert into the task network the task DESCRIPTORS for each of the subtasks in the METHOD just added.

7. Have the database machinery propagate effects through the network, and go to step 1.

This algorithm quickly fails on impossible tasks — a useful quality in a planner working in a highly exponential search space. When the algorithm is successful (which is most of the time where success is possible), an efficient plan will result. Efficiency is derived by the flexibility of least-commitment planning tempered by noticing adverse relative effects early enough to plan around them. For the problem of widget and gizmo construction discussed above, FORBIN's solution is shown in Table 1.

Task	Place	Time		
		Travel	Task	Elapsed
(get (bit widget) hand1)	bit-rack	0	1	1
(put (bit widget) hand1)	lathe-chuck	3	1	5
(push (button widget) hand1)	lathe-control	1	1	7
(wait widget)	—	0	11	—
(get (bit gizmo) hand1)	bit-rack	3	1	11
(get (bit widget) hand2)	lathe-chuck	3	1	19[a]
(put (bit gizmo) hand1)	lathe-chuck	0	1	20
(push (button gizmo) hand1)	lathe-control	1	1	22
(wait gizmo)	—	0	14	—
(get widget hand1)	lathe-hopper	3	1	26
(put widget hand1)	(shelf widget)	5	1	32[b]
(get gizmo hand1)	lathe-hopper	5	1	38[c]
(put (bit widget) hand2)	bit-rack	3	1	42
(put gizmo hand1)	(shelf gizmo)	3	1	46[d]

[a] end (wait widget) at 18

[b] end (make widget)

[c] end (wait gizmo) at 36

[d] end (make gizmo)

Table 1: FORBIN's Solution to the Example Problem

7 Contributions

This paper makes two basic types of contribution. First, FORBIN extends the state of the art in terms of representational power (*e.g.*, reasoning about space, time, and continuously changing quantities), and methods for coping with complexity (*e.g.*, the two-stage process for handling temporal projection). The second type of contribution made by this paper stems from our observations concerning the very enterprise of designing hierarchical planners and what directions that enterprise should be taking. In the following two subsections, we will consider each of the two types of contribution in turn.

7.1 Extending the State of the Art in Hierarchical Planning

FORBIN employs a number of sophisticated methods for representing and reasoning about application domains. The FORBIN plan language makes it possible to reason about travel time, resource usage, and continuously changing quantities. The language for expressing causal theories can encode rules about simultaneous events and actions whose effects depend upon the context in which they are executed. Our use of causal theories and plan libraries constitutes a significant improvement over traditional operator languages. Causal theories and plan libraries serve to partition the planner's knowledge into knowledge about the consequences of acting and knowledge about how to act. This partitioning makes it significantly simpler to maintain a large knowledge base. The details concerning the representations used in FORBIN are provided in Section 4 (causal theories) and Section 5 (plan libraries).

In addition to the representational advances, there are two technical innovations that relate to dealing with complexity that should be reiterated. The first concerns the two-stage decision process involving the use of the temporal database and the heuristic task scheduler, and the second concerns the techniques used for supporting hierarchical planning.

There are two primary components for performing inference in the FORBIN planner: the temporal database manager (specifically the query processing routines) and the heuristic task scheduler (specifically the methods for searching through simulations). Each of the two components assist in the decision process by restricting the total number of hypotheses considered at particular points during planning. The primary decisions made by the planner concern how to carry out certain tasks

(how to reduce a given nonprimitive task to subtasks) and how to schedule tasks that the planner has already committed to. In order to provide useful assistance, each of these components performs a certain amount of inference. Because of the large amount of data involved and the complexity of the basic inference problems, the answers provided by the temporal database are not guaranteed to be accurate. The inference methods employed by the temporal database manager are incomplete and, given the heuristics used, potentially unsound in certain cases. However, these inference methods are reasonably fast (even in the present prototype implementation) and generally provide accurate answers. The task expander makes use of the temporal database to initially narrow its search. The task expander asks the temporal database a set of questions concerning the applicability of various methods for carrying out the task it is currently working on. The temporal database manager responds by indicating a small number of times during which those conditions are met. The plans themselves encode information about how long the plan will take and what resources are required, and this information is used to restrict attention to particular intervals of times and methods appropriate for those intervals.

The heuristic task scheduler assists in the decision process by further narrowing the set of possibilities and adding more details (scheduling constraints) to the partially constructed plan. The heuristic task scheduler, like the temporal database manager, works by inferring consequences and exploring the repercussions of certain planning decisions. The heuristic task scheduler, however, works on a smaller data set, and hence it is able to make much more accurate predictions about a given course of action. The inference problem is essentially the same for both the temporal database manager and the heuristic task scheduler. The temporal database manager works from a partially ordered set of events, and attempts to do the best it can without considering total orders. The heuristic task scheduler works by actually exploring total orders and by using fairly general heuristics to restrict its attention to a small subset of the (potentially exponential) set of total orders. All of these machinations are to no avail however, if the decisions that they guide fail to be appropriate (*i.e.*, they have to be reversed). In order to guard against reversal, FORBIN is able to estimate, and thereby anticipate, the time required for executing a given plan and the resources that might be consumed or produced during execution. This ability to perform limited look ahead also appears in a second contribution of the FORBIN planner.

A second technical innovation in FORBIN concerns a specific strategy for making

hierarchical planning work in the FORBIN domain. This strategy relies upon the ability of the plan language to encode information concerning the projected duration and resource usage of abstract actions. It is in general impossible to anticipate whether or not a particular task reduction will succeed in a given context. If it were possible to quickly make such determinations, then planning would be simple. On the other hand, there are often questions that one can ask at different stages in the process of expanding a task that will serve to rule out certain options. The most obvious questions concern time and resource usage (*e.g.*, you won't be able to have a leisurely dinner on the way to the airport if your plane leaves in 20 minutes). Quite often, you can ask appropriate questions well before the plan is specified in any great detail (*e.g.*, it doesn't matter if you take a cab, a bicycle, or a private limousine you'll miss your plane if you stop for dinner on the way to the airport). The FORBIN plan language makes it possible to encode time and resource constraints at different levels of abstraction allowing FORBIN to avoid making bad choices early in the planning process. The resulting constraints are used to rule out impossible plans.

7.2 *Directions for Research in Planning*

In order to understand the following, it is necessary to understand our motivations for designing FORBIN. We are interested in building autonomous robots: robots capable of interesting behavior in complex domains; robots capable of operating flexibly in some approximation of real time. At the time that we began working on FORBIN we thought that hierarchical planning, as it was first characterized by NOAH and subsequently refined by such programs as NONLIN, DEVISER, and SIPE, was primarily concerned with the issues involved in building autonomous robots. It seems, however, that the central issues of hierarchical planning are largely peripheral to building autonomous robots. To the extent that hierarchical planning can be said to address a well-defined problem (*i.e.*, a problem for which there exists a precise account of what suffices as a solution), hierarchical planning is concerned with solving combinatorial problems. Combinatorial problems are essentially problems with so little structure to exploit that they admit to no efficient solution; the best known algorithms require an exponential amount of time in the worst case. A great deal of effort has gone into solving special cases of the more general combinatorial problems or providing heuristics that work well for a large number of the cases that

occur in practice.

Of course, hierarchical planning isn't really concerned with solving particular combinatorial problems; there is another branch of computer science dedicated to that enterprise. Hierarchical planning is concerned with building general frameworks for solving many interrelated combinatorial problems at once. FORBIN provides just such a general framework; it is particularly adept at reasoning about deadlines, travel time, and complex resources. We knew what sorts of things we wanted FORBIN to reason about, and we set about designing appropriate representations. We knew that FORBIN would have to produce suboptimal solutions on occasion. The problem was that we weren't clear on what occasions we would be willing to accept such suboptimal solutions. When FORBIN slowed to a crawl in attempting to solve relatively simple problems, we concluded that the slowdown must be due to the complexity of our spatial and temporal reasoning routines, and we sought to improve those routines. The real problem, however, was that we never really reconciled ourselves to having FORBIN be incomplete because we had no clear characterization of what it meant for FORBIN to be a success and yet be incomplete.

FORBIN is a useful framework for investigating very complex combinatorial problems involving space, time, and resources; a *framework* and not a solution—the solution to such problems must involve a great deal of domain specific knowledge as well. However, the sort of problems that FORBIN is most suited to are not of primary importance in developing autonomous robots. FORBIN is a significant step in the direction suggested by the evolution of planners over the last decade; it's just not a step in the direction we want to go.

7.3 Where Do We Go From Here?

There are really two questions addressed in this subsection: "Given the direction that FORBIN is heading, where do we go?" and "Given that we're interested in autonomous robots, what do we do instead?" The answer to the first question is fairly clear. We give up our preoccupation with generality and start looking at very specific domains. The ISIS project [9] is a good example of this approach. We admit to the difficulty of the problems we are looking at and attempt to characterize the sort of incompleteness that we are willing to accept. The SIPE planner [17] is a good example of a research effort that has dealt directly with the tradeoffs between generality and practicality.

The second question is a lot more difficult to answer. It is usually easier to determine how to extend an existing paradigm than it is to determine what to do when you abandon one. There are, however, certain issues that have not yet been addressed in the hierarchical planning paradigm and will serve to focus work in the coming years. The most important of these concerns a basic distinction between tasks that involve the control of uncertain, changing processes (*e.g.*, the coordination of actions for catching a flying object [6]) and the predictable, static tasks that have dominated planning for the last two decades. Given that most domains do not allow making detailed predictions, recognizing and responding quickly to situations is probably the most important issue determining the course of future research. Already a number of researchers have begun to suggest approaches to dealing with such control issues [1] [2] [13].

7.4 Final Word

Traditional hierarchical planning is not the solution to every problem. In particular, hierarchical planning does not directly address the control issues that dominate in designing autonomous robots. Traditional hierarchical planning is primarily concerned with complex combinatorial problems and FORBIN constitutes a solution to several of the representational problems that have limited its application in the past. In addition, FORBIN demonstrates how the prediction problem can be partitioned in a reasonable way. In order to be useful, FORBIN requires a great deal of domain specific knowledge which is exactly what one would expect given the combinatorial nature of the problems that FORBIN seeks to solve. It is our contention that any further substantive extensions to the theory of hierarchical planning will involve (i) considerable use of domain knowledge, (ii) an important heuristic component, and (iii) a meaningful characterization of the incompleteness that results from using the heuristics embedded in the system. FORBIN provides a useful framework for exploring such extensions.

References

1. Brooks, Rodney A., A Robust Layered Control System for a Mobile Robot, A.I. Memo No. 864, MIT AI Laboratory, 1985.

2. Chapman, David, and Phil Agre, Abstract Reasoning as Emergent from Concrete Activity, in Georgeff, Michael P. and Lansky, Amy L. (Eds.), *The 1986 Workshop on Reasoning about Actions and Plans*, (Morgan-Kaufman 1987).

3. Chapman, David, *Planning for Conjunctive Goals, Artificial Intelligence* **33** (1987).

4. Cheeseman, Peter, A Representation of Time for Automatic Planning, *Proceedings IEEE Int. Conf. on Robotics*, IEEE, 1984.

5. Dean, Thomas, *Temporal Imagery: An Approach to Reasoning about Time for Planning and Problem Solving*, Technical Report 433, Yale University Computer Science Department, 1985.

6. Donner, Marc D. and David H. Jameson, *A real-time juggling robot*, IBM Research Report RC 12111 (54549), 1986.

7. Ernst, G. and Allen Newell, *GPS: A Case Study in Generality and Problem Solving*, (Academic Press, New York, 1969).

8. Fikes, Richard and Nils J. Nilsson, STRIPS: A new approach to the application of theorem proving to problem solving, *Artificial Intelligence* **2** (1971) 189–208.

9. Fox, M.S., and Stephen Smith, ISIS: A Knowledge-Based System for Factory Scheduling, *Expert Systems* **1** (1984) 25–49.

10. McDermott, Drew V., *Flexibility and Efficiency in a Computer Program for Designing Circuits*, Technical Report 402, MIT AI Laboratory, 1977.

11. McDermott, Drew V., *The DUCK Manual*, Technical Report 399, Yale University Computer Science Department, 1985.

12. Miller, David P., *Planning by Search Through Simulations*, Technical Report 423, Yale University Computer Science Department, 1985.

13. Rosenschein, Stanley J. and Leslie Pack Kaelbling, The Synthesis of Digital Machines with Provable Epistemic Properties, in Halpern, Joseph Y. (Ed.), *Theoretical Aspects of Reasoning about Knowledge, Proceedings of the 1986 Conference*, (Morgan-Kaufman 1987).

14. Sacerdoti, Earl, *A Structure for Plans and Behavior* (American Elsevier Publishing Company, Inc., 1977).

15. Tate, Austin, Generating Project Networks, *Proceedings IJCAI 5, Cambridge, Ma.*, IJCAI, 1977.

16. Vere, Steven, Planning in Time: Windows and Durations for Activities and Goals, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **5** (1983) 246–267.

17. Wilkins, David E., Recovering from execution errors in SIPE, *Computational Intelligence*, **1(1)**, (February 1985), 33–45.

A The FORBIN Factory Domain

The FORBIN planner is a general purpose system, but for illustration we will present examples from a small automated factory domain. The factory consists of an automatic lathe, a storage area and a robot operator that builds *widgets* and *gizmos* and places them in the storage area (see Figure 18). Making either a widget or gizmo requires the robot to carry out the following steps:

1. Make sure the correct bit is in the lathe.
2. Start the lathe and let it run until finished (11 units of time for a gizmo and 14 units for a widget).
3. Remove the finished item from the lathe hopper.
4. Place the item on the correct shelf.

There are three primitive actions, **get**, **put**, **push** each taking one unit of time to complete and requiring the use of one of the robot's hands. The robot has two hands, each of which can carry one item or be used to operate a control. The robot can travel throughout the factory at a fixed velocity and the travel times between the various work stations are given in Table 2. A typical problem is to construct a number of widgets and gizmos where some of each must be done within specific deadlines.

This factory differs from typical job-shop factories like those handled by the ISIS program [9] in two important respects. First, a complete "job" does not travel from workstation to workstation as a single entity (eg., not all of the widgets in an order need to be turned on the lathe before any can be moved to the storage shelves) and second, the travel time of the robot from one place in the factory to another can be a significant part of the overall factory production time.

This domain illustrates the importance of temporal representation and plan efficiency issues that have not been adequately handled by previous planning systems.

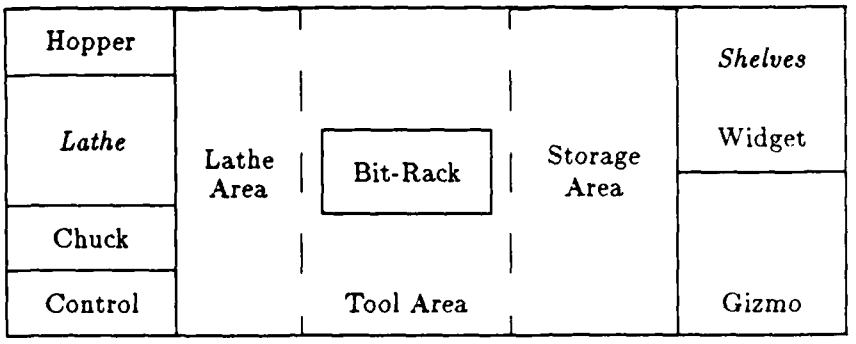


Figure 18: Layout of the Factory

	Hopper	Chuck	Control	Rack	Widget	Gizmo
Hopper	0	2	3	3	5	6
Chuck	2	0	1	3	5	5
Control	3	1	0	3	6	5
Rack	3	3	3	0	3	3
Widget	5	5	6	3	0	2
Gizmo	6	5	5	3	2	0

Table 2: Travel Times in the Factory

B The FORBIN Factory Task Expansion Library

This appendix contains all of the task expansion methods referenced in Section 6 of the text. These methods are supplied for the sake of completeness. The world physics and task descriptors for this domain are supplied in appendix C.

```
; -- (MAKE ?TYPE) --

(DEFINE-METHOD MAKE
   (TASK (MAKE ?TYPE))
   (UTILITY  1.0)
   (ASSUMPTIONS nil)
   (SUBTASKS
      (T1 (GENERATE ?TYPE $NEW-THING)
          (CREATE  ?TYPE $NEW-THING) FOR T2)
      (T2 (MOVE $NEW-THING (LOC-OF SHELF ?TYPE))
          (ACHIEVE (LOCATION $NEW-THING (LOC-OF SHELF ?TYPE))))))


(DEFINE-METHOD GENERATE
   (TASK (GENERATE ?TYPE ?THING))
   (UTILITY  1.0)
   (ASSUMPTIONS
      (RESERVE (BEGIN *SELF*) (END *SELF*) LATHE ?LATHE (NAME *SELF*)))
   (SUBTASKS
      (T1 (SETUP-LATHE ?LATHE ?TYPE)
          (ACHIEVE (READY ?LATHE ?TYPE)) FOR T2)
      (T2 (RUN-LATHE ?LATHE ?TYPE)
          (CREATE  ?TYPE ?THING)
          (ACHIEVE (LOCATION ?THING (LOC-OF HOPPER ?LATHE))))))
```

```
; -- (SETUP-LATHE ?LATHE ?TYPE) --

(DEFINE-METHOD SETUP-LATHE
   (TASK (SETUP-LATHE ?LATHE ?TYPE))
   (UTILITY  1.0)
   (ASSUMPTIONS
      (TT (BEGIN *SELF*) (BEGIN *SELF*)
          (LOCATION (BIT-FOR ?TYPE) ?LOC1)))
   (SUBTASKS
      (T1 (GET (BIT-FOR ?TYPE) ?LOC1)
          (ACHIEVE (HAVE *ME* (BIT-FOR ?TYPE))) <FOR T2)
      (T2 (INSTALL-BIT (BIT-FOR ?TYPE) ?LATHE)
          (ACHIEVE (READY ?LATHE ?TYPE)))))

; -- (RUN-LATHE ?LATHE ?TYPE) --

(DEFINE-METHOD RUN-LATHE
   (TASK (RUN-LATHE ?LATHE ?TYPE))
   (UTILITY  1.0)
   (ASSUMPTIONS nil)
   (SUBTASKS
      (T1 (PUSH (BUTTON ?TYPE) (LOC-OF CONTROL ?LATHE)) FOR T2)
      (T2 (WAIT (TIME ?TYPE)))))


; -- (INSTALL-BIT ?BIT ?LATHE) --

(DEFINE-METHOD INSTALL-1
   (TASK (INSTALL-BIT ?BIT ?LATHE))
   (UTILITY 2.0)
   (ASSUMPTIONS
      (TT (BEGIN *SELF*) (BEGIN *SELF*) (BIT ?LATHE ?OTHER-BIT))
      (NOT ?OTHER-BIT BIT)
      (NOT ?OTHER-BIT NONE))
   (SUBTASKS
      (T1 (REMOVE-BIT ?OTHER-BIT ?LATHE)
          (ACHIEVE (BIT ?LATHE NONE)) <FOR T2)
      (T2 (PUT ?BIT (LOC-OF CHUCK ?LATHE))
          (ACHIEVE (BIT ?LATHE ?BIT)))))
```

```
(DEFINE-METHOD INSTALL-2
   (TASK (INSTALL-BIT ?BIT ?LATHE))
   (UTILITY 1.0)
   (ASSUMPTIONS
      (TT (BEGIN *SELF*) (BEGIN *SELF*) (BIT ?LATHE NONE)))
   (SUBTASKS
      (T1 (PUT ?BIT (LOC-OF CHUCK ?LATHE))
          (ACHIEVE (BIT ?LATHE ?BIT)))))

; -- (REMOVE-BIT ?BIT ?LATHE) --

(DEFINE-METHOD REMOVE-BIT
   (TASK (REMOVE-BIT (BIT-FOR ?TYPE) ?LATHE))
   (UTILITY 1.0)
   (ASSUMPTIONS nil)
   (SUBTASKS
      (T1 (GET (BIT-FOR ?TYPE) (LOC-OF CHUCK ?LATHE))
          (ACHIEVE (HAVE *ME* ?BIT)) FOR T2)
      (T2 (PUT (BIT-FOR ?TYPE) (LOC-OF BIT-RACK ?TYPE))
          (ACHIEVE (LOCATION (BIT-FOR ?TYPE)
                             (LOC-OF BIT-RACK ?TYPE))) FOR>)))

; -- (MOVE ?THING ?FINISH)

(DEFINE-METHOD MOVE
   (TASK (MOVE ?THING ?FINISH))
   (UTILITY  1.0)
   (ASSUMPTIONS
      (TT (BEGIN *SELF*) (BEGIN *SELF*) (LOCATION ?THING ?START)))
   (SUBTASKS
      (T1 (GET ?THING ?START)
          (ACHIEVE (HAVE *ME* ?THING)) FOR T2)
      (T2 (PUT ?THING ?FINISH)
          (ACHIEVE (LOCATION ?THING ?FINISH)))))
```

C The FORBIN Factory Causal Theory

This appendix contains the causal theory declarations necessary to complete the domain specific knowledge FORBIN uses in the factory domain.

```
; Fact declarations:

(DECLARE-FACT (BIT ?lathe ?bit))
(DECLARE-FACT (READY ?lathe ?type))
(DECLARE-FACT (LOCATION ?thing ?place))
(DECLARE-FACT (HAVE ?robot ?thing))

; CLipping rules:

(DECLARE-CLIP (BIT ?lathe ?bit) (BIT ?lathe ?other)
              (NOT (= ?bit ?other)))
(DECLARE-CLIP (READY ?lather ?type1) (READY ?lathe ?type2)
              (NOT (= ?type1 ?type2)))
(DECLARE-CLIP (LOCATION ?thing ?place1) (LOCATION ?thing ?place2)
              (NOT (= ?place1 ?place2)))
(DECLARE-CLIP (HAVE ?robot ?thing) (LOCATION ?thing ?place)
              (NOT (= ?robot ?place)))

; State declarations:

(DECLARE-STATE POSITION
   (MOVE   (LAMBDA (ORIG DEST TIME) T))
   (DELAY  (LAMBDA (ORIG DEST TIME)
               (LOOKUP-TRAVEL-TIME ORIG DEST)))
   (UPDATE (LAMBDA (ORIG DEST TIME)
               (ACTION (ROBOT-MOVE-TO DEST))
               DEST)))

; Pools of objects to be referenced:

(DECLARE-POOL LATHE  (LATHE-A LATHE-B))
(DECLARE-POOL WIDGET ())
(DECLARE-POOL GIZMO  ())
```

```
; Task Descriptors:

(TASK-DESCRIPTOR
   (TASK (MAKE ?TYPE))
   (ESTIMATED-DURATION 21.0 37.0)
   (EXPECTED-STATES
      ())
   (EXPECTED-FACTS
      ()))

(TASK-DESCRIPTOR
   (TASK (GENERATE ?TYPE ?NAME))
   (ESTIMATED-DURATION 17.0 31.0)
   (EXPECTED-STATES
      ())
   (EXPECTED-FACTS
      ()))

(TASK-DESCRIPTOR
   (TASK (SETUP-LATHE ?TYPE))
   (ESTIMATED-DURATION 5.0 6.0)
   (EXPECTED-STATES
      (END (POSITION (CHUCK ?LATHE))))
   (EXPECTED-FACTS
      (READY ?LATHE ?TYPE)
      (BIT ?LATHE (BIT-FOR ?TYPE))
      (LOCATION *ME* (LOC-OF CHUCK ?LATHE))))

(TASK-DESCRIPTOR
   (TASK (INSTALL-BIT ?BIT ?LATHE))
   (ESTIMATED-DURATION 1.0 2.0)
   (EXPECTED-STATES
      ())
   (EXPECTED-FACTS
      (BIT ?LATHE ?BIT)))
```

```
(TASK-DESCRIPTOR
   (TASK (REMOVE-BIT ?BIT ?LATHE))
   (ESTIMATED-DURATION 1.0 2.0)
   (EXPECTED-STATES
      ())
   (EXPECTED-FACTS
      ()))

(TASK-DESCRIPTOR
   (TASK (RUN-LATHE ?LATHE ?TYPE))
   (ESTIMATED-DURATION 12.0 14.0)
   (EXPECTED-STATES
      (START (POSITION (CONTROL ?LATHE))))
   (EXPECTED-FACTS
      ()))

(TASK-DESCRIPTOR
   (TASK (MOVE ?THING ?FINISH))
   (ESTIMATED-DURATION 7.0 8.0)
   (EXPECTED-STATES
      (START (POSITION (HOPPER ?LATHE)))
      (END   (POSITION (SHELF  ?TYPE))))
   (EXPECTED-FACTS
      (LOCATION ?THING ?FINISH)
      (LOCATION *ME* ?FINISH)))

; -- PRIMITIVES --

(TASK-DESCRIPTOR
   (TASK-PRIMITIVE (GET ?THING ?PLACE))
   (ESTIMATED-DURATION 1.0 1.0)
   (EXPECTED-STATES
      (START (POSITION ?PLACE)))
   (EXPECTED-FACTS
      (HAVE *ME* ?THING)
      (LOCATION *ME* ?PLACE)))
```

```
(TASK-DESCRIPTOR
   (TASK-PRIMITIVE (PUT ?THING ?PLACE))
   (ESTIMATED-DURATION 1.0 1.0)
   (EXPECTED-STATES
      (START (POSITION ?PLACE)))
   (EXPECTED-FACTS
      (LOCATION ?THING ?PLACE)
      (LOCATION *ME* ?PLACE)))

(TASK-DESCRIPTOR
   (TASK-PRIMITIVE (PUSH ?BUTTON ?PLACE))
   (ESTIMATED-DURATION 1.0 1.0)
   (EXPECTED-STATES
      (START (POSITION ?PLACE)))
   (EXPECTED-FACTS
      (LOCATION *ME* ?PLACE)))

(TASK-DESCRIPTOR
   (TASK-PRIMITIVE (WAIT ?TIME))
   (ESTIMATED-DURATION ?TIME ?TIME)
   (EXPECTED-STATES
      ())
   (EXPECTED-FACTS
      ()))
```

END

9-87

DTIC